ICT IP Project

Deliverable D1.3

Initial Architectural Style for CHOReOS Choreographies

http://www.choreos.eu

LaTeX template v. 1.14

Project Number	:	FP7-257178
Project Title	:	CHOReOS Large Scale Choreographies for the Future Internet

Deliverable Number	:	D1.3
Title of Deliverable	:	Initial Architectural Style for CHOReOS Choreographies
Nature of Deliverable	:	Report
Dissemination level	:	Public
Licence	:	Creative Commons Attribution 3.0 License
Version	:	0.1
Contractual Delivery Date	:	30 September 2011
Actual Delivery Date	:	14 October 2011
Contributing WP	:	WP1
Editor(s)	:	Valérie Issarny (INRIA)
Author(s)	:	Nikolaos Georgantas (INRIA), Sara Hachem (INRIA), Valérie Issarny (INRIA), Marco Autili (UDA), Davide Di Ruscio (UDA), Paola Inverardi (UDA), Massimo Tivoli (UDA), Dionysis Athanasopoulos (UOI), Panos Vasilliadis (UOI), Apostolos Zarras (UOI), Daniel Batista (USP), Carlos Eduardo Moreira dos Santos (USP)
Reviewer(s)	:	Nikolaos Georgantas (INRIA), Jean-Pierre Lorré (EBM), Hugues Vincent (THALES)

Abstract

While the development of CHOReOS systems build on well-known paradigms associated with service-oriented architectures (e.g., services, service bus and service choreography), the supporting architectural style requires accounting for the challenges posed by the future Internet, i.e., ultra large scale, high heterogeneity, increased mobility, and awareness & adaptability. This deliverable then revisits the traditional definitions of service-oriented *component* (i.e., *service*), *connector* (*interaction protocol* and related *service bus for interoperability*) and *configuration* (*system-wide architecture composing services* according to orchestration or more general choreography patterns) to meet the FI challenges. Specifically, CHOReOS components enable leveraging the diversity of Web-based services that integrate in the FI (i.e., WS$*$ and RESTful web-based services, and from business to thing-based services) as well as the ultra large service base envisioned for the FI. As for CHOReOS connectors, they bring together the highly heterogeneous interaction paradigms that are now used in today's increasingly complex distributed systems and further support interoperability across heterogeneous paradigms. Finally, CHOReOS coordination protocols foster choreography-based coordination for the sake of scalability, while preventing undesired behavior (i.e., undesired service interactions that would violate the specified choreography). A key aspect of the proposed CHOReOS architectural style is to introduce novel abstractions for all its elements, which enable leveraging the wide diversity of the FI, in all the dimensions of scale, heterogeneity and mobility. The CHOReOS style further sets the base ground for the development (from design to implementation) of the CHOReOS Integrated Development and Runtime Environment, and especially for the specification and design of choreography-based systems (studied in WP2 complemented with WP4 work on Governance and V&V) and the development of the CHOReOS service-oriented middleware (studied in WP3).

Keyword List

Architectural Style, Components, Connectors, Coordination, Choreography, Future Internet, Interaction paradigms, Scalability, Interoperability

Document History

Version	Changes	Author(s)
0.1	Outline Draft	Valérie Issarny (INRIA)
1.x	First version and revision of individual chapters	All authors
2.0	Overall integration and edition	Valérie Issarny (INRIA)
3.0	Final revision	Valérie Issarny (INRIA)

Document Reviews

Review	Date	Ver.	Reviewers	Comments
Outline	13 May 2011	1.0	All authors	Outline agreed by all
Draft	30 Sept. 2011	1.x	Valérie Issarny	Intermediate version released for final review
QA	7 Oct. 2011	2.0	Nikolaos Georgantas, Jean-Pierre Lorré, Hugues Vincent	Editorial comments
PTC	12 Oct. 2011	A	PTC	-

Glossary, acronyms & abbreviations

Item	Description
BPEL	Business Process Execution Language
BPEL4SWS	Business Process Execution Language for Semantic Web Services
BPMN	Business Process Modeling Notation
BPMN2	Business Process Modeling Notation - ver. 2
CBA	Choreography Based Architecture
CCS	Calculus of Communicating Systems
CS	Client Server
CSP	Communicating Sequential Processes
DAML	DARPA Agent Markup Language
DAML-S	DARPA Agent Markup Language for Services
DOW	Description of Work
ESB	Enterprise Service Bus
FI	Future Internet
GA	Generic Application
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
IDRE	Integrated Development and Runtime Environment
IOPE	Inputs, Outputs, Preconditions, Effects
JBI	Java Business Integration
JSON	JavaScript Object Notation
LTS	Labeled Transition System
NEXOF	NESSI Open Framework
OASIS	Organization for the Advancement of Structured Information Standards
OWL	Ontology Web Language
OWL-S	Ontology Web Language for Services
PaaS	Platform as a Service
PS	Publish Subscribe
REST	REpresentational State Transfer
S & A	Sensor and Actuators
SaaS	Software as a Service
SAWSDL	Semantically Annotated Web Service Description Language
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOM	Service Oriented Middleware
TS	Tuple Space
ULS	Ultra-Large-Scale
ULS-FI	Ultra-Large-Scale Future Internet
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WADL	Web Applications Description Language
WP	Work Package
WSCL	Web Service Conversation Language
WSDL	Web Service Definition Language
WSDL-S	Web Service Description Language with Semantics
WSN	Wireless Sensor Network
WSAN	Wireless Sensor and Actuator Network

WSCL	Web Service Conversation Language
WSQM	Web Service Quality Model
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

Table Of Contents

List Of Tables

List Of Figures

1 Introduction

A *software architecture style* characterizes the types of: *components* (i.e., units of computation or data stores), *connectors* (i.e., interaction protocols) and possibly *configurations* (i.e., system structures) that serve building a given class of system. As such, the definition of a software architectural style is central toward eliciting appropriate development and runtime support for any family of systems. Indeed, the style elements altogether specify the abstractions that need to be modeled, from design to implementation, as well as supported by the runtime to enact the target systems.

Within the CHOReOS project, we adopt the *Service*, and related *Service Oriented Architecture* (SOA) and *Service Computing* paradigms for the Future Internet (FI). In other words, networked systems of the FI are exposed as (software) service providers and/or consumers (with service *prosumer* standing for a service that acts both as a consumer and a provider) in the networking environment. The service abstraction specifically serves characterizing functionalities that are provided and required in the networked environment so as to enable dynamic lookup and further binding between matching service consumer and producer. The service oriented architecture style may then be briefly defined as:

1) Components map to services, which may be refined into consumer, producer or prosumer services

2) Connectors map to traditional client-service interaction protocols, and

3) Configurations map to compositions of services through (service-oriented) connectors, i.e., choreography in the most general form, and orchestration as a specific composition structure that is commonly adopted in today's Internet.

The service-oriented architectural style, as defined above, has led to many refinements, and further development of associated middleware technologies. Then, acknowledging the diversity of services to be composed, the service bus paradigm (*aka* ESB - Enterprise Service Bus, with the Enterprise qualifier stressing the emergence of the paradigm from the Information system domains) has been largely adopted in SOA, thereby introducing a connector type oriented toward interoperability across heterogeneous service-oriented middleware.

While, the service-oriented architecture style is well suited to support the development of Internet-based distributed systems, it is largely challenged by the Future Internet that poses new demands in terms of sustaining the following *ities* [77][1]:

- *Scalability:* The service abstraction shall serve characterizing and further locating all the component systems that will get networked at a massive scale in the Future Internet. Scalability further questions the so-far dominating, strongly coupled client-server paradigm for service-oriented systems and advocates for weakly coupled interactions. As for structural concerns, centralization shall be the exception, hence calling for choreography-based service composition.

- *Heterogeneity:* Service abstractions are already much heterogeneous in today's Internet, where both WS* and RESTful services co-exist. They will be even more so as highly heterogeneous

[1] While enforcing security, privacy and trust in the Future Internet is another key challenge identified in [77], it is not considered in this document, as it is beyond the scope of CHOReOS work.

services, ranging from Business to Thing-based ones, will be integrated within the Future Internet. The heterogeneity of networked services further lead to account for various connector types implementing different interaction paradigms, from strongly to weakly coupled, as well as for interoperability across interaction paradigms. Heterogeneity has fewer impact on the composition of services, as it is to be abstracted at the service and connector levels.

- *Mobility:* Mobility directly affects all the architectural elements, as it calls for system architectures whose components may be highly volatile and hence requires dealing with service substitution within composition. Architecture reconfiguration shall in particular be supported by appropriate service abstractions and weakly coupled interaction paradigms.

- *Awareness & adaptability:* This challenge relates to the above ones in terms of impact on the architectural elements, especially regarding supporting service abstraction and weakly coupled interactions. In addition, at the structural level, it must be considered that services that get composed have not been designed in conjunction and thus may exhibit erroneous behavior when choreographed.

The next three chapters introduce the CHOReOS software architecture style that we have elaborated to address the above challenges. Specifically:

- Chapter 2 defines CHOReOS services in a way that is technology-agnostic and allows for abstracting Business as well as Thing-based services. The main innovation of the chapter comes from the definition of new *functional and non-functional abstractions* for services that enables hierarchical structuring and hence scalable *abstraction-oriented service bases*. Proposed abstractions also ease service substitution at runtime as service abstractions define pivot services against which behavior and interface of functionally matching service instances may be adapted.

- Chapter 3 tackles the issue of interoperability across heterogeneous interaction paradigms, from strongly to weakly coupled ones. It does so through the introduction of: (i) *base connector types* abstracting core interaction paradigms implemented by state of the art middleware solutions, and (ii) the *Generic Application - GA connector type* that realizes interoperability in a way similar to a service bus, but across interaction paradigms. The main contribution of the chapter lies in enabling cross-paradigm interoperability, while preserving the behavioral (at middleware-level) semantics of connected components.

- Chapter 4 focuses on the formalization of the notion of CHOReOS *coordination protocol* that abstracts choreography behavior. The chapter specifically defines architectural constraints to be imposed on the choreography-based system to suitably coordinate the composed services, and further enable the *automated synthesis* of the coordination protocol. The main novelty of the chapter derives from the efficient production of a *decentralized choreographer*, while avoiding undesired service interactions, where the undesired interactions are those interactions that can happen by letting the discovered participant services collaborate in an uncontrolled way and that do not belong to the set of interactions modeled by the choreography specification.

Chapter 5 finally concludes with our ongoing and future work toward the final definition of the CHOReOS architectural style. This work will in particular be informed by the concrete development of the CHOReOS IDRE whose elements are designed based on the initial architectural style introduced in this document.

2 CHOReOS Components: Abstracting Services

As reported in Deliverable D1.2 [77], the CHOReOS vision is to sustain choreography-based service composition in the Future Internet of Services and Smart Things, while addressing the *scalability*, the *heterogeneity*, the *mobility*, and the *awareness & adaptability* challenges introduced in this context (see also the previous section). The specific impact of these high-level challenges from the perspective of the CHOReOS components, i.e., the services used for the development of CHOReOS choreographies, may be summarized as follows:

- *Scalability:* The Future Internet scale-up prompts to the issue of easily exploiting the constantly growing amount of services that provide similar functional/non-functional properties. This issue relates to both service discovery and the composition of services within choreographies.
- *Heterogeneity:* Concerning heterogeneity, a main problem that we expect to face in the Future Internet is the diversity of the service-related standards and technologies assumed by the two dominant paradigms of the Internet of Services, namely WS* services [83] and Restful services [30] that both target Web-based services. Further, the FI aggregates highly heterogeneous services from various domains, and especially the Business and Thing domains, as investigated within CHOReOS. Specifically, CHOReOS concentrates on a Future Internet of heterogeneous (Web-based) services that integrates Business and Thing-based services.
- *Mobility:* Mobility mainly relates to the envisioned global-scale usage of increasingly smarter mobile devices, which are characterized by certain mobility-related requirements and resource constraints (see Deliverable D1.2 [77]).
- *Awareness & Adaptability:* Finally, the growing amount and the diversity of available services further prompts to the need for building choreographies that are aware of what is out there in the Future Internet and choreographies that are capable to easily adapt with respect to their possibly evolving requirements and the evolving properties of the services that become available over time.

Dealing with the impact of the Future Internet challenges on the CHOReOS components is one of the main issues dealt with in the CHOReOS architectural style. In particular, the CHOReOS architectural style contribution to this direction is summarized in the following points:

- Regarding *heterogeneity* and *mobility*, the CHOReOS architectural style provides *a unified formal definition of services* that abstracts details related to the particular paradigms, standards and technologies on which services are based and further accounts for the specificities of Things-based services. The overall CHOReOS development process, methods, tools and middleware rely on this general perspective on services and therefore are paradigm/standard/technology independent in the sense that their core elements can be easily customized to support any type of services. The CHOReOS definition of services is inspired by previous attempts that aimed at a paradigm/standard/technology independent definition of services. The most recent attempt was the NEXOF reference architecture [53] which served as a basis for the CHOReOS conceptual model [77]. However, the NEXOF definition of services is rather informal. Moreover it does not account for certain core aspects of services such as the services' behavior. Certain other core aspects, such as the services non-functional properties, are not clearly addressed.

- Considering *scalability, awareness & adaptability*, the CHOReOS architectural style formally defines the concept of service abstractions, which was informally introduced in D1.2. Service abstractions represent groups of alternative services that provide similar functional/non-functional properties through different interfaces. The service abstractions provide unified abstract interfaces and mappings between the abstract interfaces and the interfaces of the represented services. The service abstractions are first-class building blocks in the CHOReOS development process supporting the discovery of services, the choreography-based service composition and adaptation. Concerning service discovery, CHOReOS employs abstractions to facilitate searching and browsing the vast amount of services that become available over time. On the other hand, the CHOReOS choreographies synthesis relies on service abstractions, instead of being based on concrete services. Consequently, the adaptation of choreographies, realized in terms of service substitutions becomes straightforward. The CHOReOS definition of service abstractions is inspired by previous attempts that aimed at the semantic description of services such as DAML-S [25] and its successor OWL-S [81]. These attempts proposed corresponding notations that allow specifying groups of services characterized by a common semantic description. However, the CHOReOS style takes one step further since the CHOReOS service abstractions can be hierarchically structured. Moreover, the CHOReOS style considers two different types of interrelated service abstractions: functional abstractions that represent groups of services that offer similar functional properties and non-functional abstractions that represent groups of services that provide similar non-functional properties.

Based on the previous discussion the remainder of this chapter is structured as follows. First we focus on heterogeneity and mobility issues (Section 2.1), which leads us to provide the CHOReOS definitions of Web-based services in general (Section 2.1.1), and Things-based services (Section 2.1.2) in particular, from which results the formal definition of CHOReOS components (Section 2.1.3). Then, we concentrate on the formal definitions of the CHOReOS service abstractions (Section 2.2), which address the issues of scalability and awareness & adaptability.

2.1. Services in the FI

From a software architecture point of view, the main features of components include the components' *interfaces, semantics, constraints* and *non-functional properties* [47]. An interface specifies a set of interaction points between a component and the component's environment. The semantics specify the component's behavior. The constraints refer to properties or assertions that must be satisfied so that the component operates correctly. Finally, the component's non-functional properties may comprise various features related to the component's performance, reliability, availability, etc. In line with this perspective, in CHOReOS we also consider these main features as the basic constituents of the CHOReOS definition of services. Still, such a definition shall account for the heterogeneity of services to be aggregated in the FI, while acknowledging that services are essentially (if not uniquely) Web-based at the (Future) Internet level although Web-based services are called to evolve to face the FI challenges (e.g., see next chapter on the need to deal with diverse interaction paradigms).

2.1.1. Web-based services (WS* & RESTful)

To come up with a unified definition of Web-based services that deals with the heterogeneity derived from the two main service-oriented paradigms, i.e., WS* and RESTful, and associated standards and technologies, we surveyed these two main paradigms. The goal of this task was to recall the specificities of these two service-oriented paradigms, concerning the main features that generally characterize components, as seen from the software architecture point of view.

WS and RESTful services: Brief definitions*

WS*: According to the W3C standards [83]: "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using (traditionally) SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

RESTful: According to [66]: "A RESTful Web service (also called a RESTful Web API) is a simple Web service implemented using HTTP and the principles of REST. It is a collection of resources, with three defined aspects: the base URI for the Web service; the Internet media type of the data supported by the Web service (this is often JSON, XML or YAML but can be any other valid Internet media type); the set of operations supported by the Web service using HTTP methods (e.g., POST, GET, PUT or DELETE)."

Service interface

WS*: The most widely known standard for the specification of WS* interfaces is WSDL[1]. Although the existence of a standard interface specification language usually helps towards establishing a unified way for defining and using WS* service interfaces, the subsequent versions of WSDL that were proposed over time have significant differences, which raise further heterogeneity issues that should be handled.

In particular, WSDLv1.1 is the older version of the language and the most widely adopted. According to this version, an interface is a set of operations. Each operation accepts as input at most one message and produces as output at most one message. Moreover, an operation may produce zero or more fault messages. A message is hierarchically structured, consisting of a set of parts. Each part may be characterized by an XML built-in type, or by an XML complex type, which in turn consists of further built-in or complex elements. Although an operation is not explicitly associated with a particular message exchange pattern, the standard distinguishes 4 different types of operations from that standpoint: *One-way*, *Request-response*, *Solicit-response*, and *Notification*, whose respective semantics are obvious from the employed names.

WSDLv2 is the latest version of the language. According to this version an interface is a set of operations. Each operation may accept as input multiple messages and produce as output multiple messages. As in the case of WSDLv1.1, an operation may further produce zero or more fault messages. Intuitively, having an operation that can accept multiple inputs and produce multiple outputs, is equivalent to having a set of overloaded operations with at most one input/output message. An operation is explicitly associated with a message exchange pattern that defines the sequence and cardinality of messages sent and/or received, as well as the entities they are logically sent to and/or received from. The standard exchange patterns are 2, involving a single and a pair of messages, respectively: *One-way* and *Request-response*.

In both versions, an interface may be associated with multiple bindings, which specify concrete message formats and transmission protocols which may be used to access the service. Typically, the message format relies on SOAP [84], while the protocol is HTTP. However, nothing prohibits from using any other format or protocol.

Apart from the different versions of WSDL, a promising work in progress for the specification of services is the Unified Service Description Language (USDL) [89], which focuses on the business aspects of services.

RESTful: In the case of RESTful services there is no *de facto* standard for specifying service interfaces. Typically, the specifications may be informal, in a textual format, or more formal given in terms

[1] http://www.w3.org/TR/wsdl.

of an XML schema. WSDL may also be used [61]. However, recently there is a proposal for a Web Applications Description Languages (WADL) [88], which concentrates mainly on RESTful services. According to this language, a service corresponds to a resource. The resource definition consists of a set of methods (i.e., operations in WSDL terms) that represent basic functionalities that can be used for accessing the resource. A method is characterized by an identifier and is associated with the HTTP primitive used for accessing the service (e.g., PUT, POST, GET). Moreover the method is associated with an input message and zero or more output messages. A message may have multiple alternative representations, where each representation may be based on XML or any other format (e.g., JSON, YAML or any other valid Internet media type).

Service semantics, behavior & constraints

WS*: A significant standardization effort for the specification of WS* service semantics is OWL-S [81]. In OWL-S, a semantic service description is composed of three parts: the service profile, which is complemented by the process model and the service grounding. The service profile provides semantic descriptions of the service's capabilities and constraints (the OWL-S term for service's high-level functionalities) in terms of Inputs, Outputs, Preconditions and Effects; these are collectively denoted as IOPEs. Inputs specify the data required by the service for its execution and Outputs specify the data provided by the service as result of its execution. In addition, Preconditions need to be fulfilled before the service may execute, while Effects specify the impact of the service on the state of the world besides its Outputs. The process model specifies service behavior in terms of processes. Atomic processes correspond to primitive operations performed by a service, while it is also possible to specify complex processes that combine other processes. The service grounding maps a semantic service specification to a concrete syntactic service specification, such as WSDL. Other efforts for semantic service specification include the WSMO initiative [41] that provides an ontology-based framework for the discovery and usage of web services, mainly exploiting a semantic annotation in the form of an ontology (also expressed via the WSMO-Lite [24] framework). OWL-S and WSMO were candidates for becoming the W3C standard for semantic service description; however, the winner was SAWSDL [82].

SAWSDL [82] is the W3C Recommendation for adding semantic annotations to WSDL and XML schemas. Such annotations can be expressed in any ontology language, most often in OWL. Annotations can be added to WSDL interfaces, operations, and the XML Schema types of their input/output messages. Moreover, SAWSDL supports introducing two-way transformation mappings between XML Schema types and corresponding semantic concepts. This enables interoperating between syntactically mismatching input/output messages upon service invocation.

Further means that allow specifying the behavior of WS* are the Web Service Conversation Language (WSCL) [85] and the Business Process Execution Language (BPEL) [56]. According to WSCL the behavior of a service is specified as a sort of a transition system. The basic modeling elements are interactions, which refer to messages exchanges between the service and the environment. Then, transitions are used to specify subsequent interactions that may take place between the service and the environment. On the other hand, BPEL has been used to specify service behavior in terms of a process model consisting of primitive and complex activities. The process specification further includes the description of fault and event handlers. Complex activities prescribe the order of the execution of their constituent activities. In other words, they define control and data flow dependencies. The different kinds of basic activities supported are: (1) invoke activities, specifying the synchronous, or asynchronous invocation of a service operation ; (2) receive activities, describing the reception of request messages that initiate a process; (3) reply activities, delineating responses to request messages that were previously received during the execution of receive activities; (4) different kinds of structured activities that are traditional programming language statements (i.e., sequence, switch, while, and pick consisting of one or more event handlers); (5) flow activities, comprising one or more activities, which by default execute concurrently - although, there may exist control and data flow dependencies between the activities, imposing a certain execution order. In USDL [89], the participants and the interaction

modules shall be provided to support the behavioral specifications of services.

RESTful: Concerning the RESTful services, there exist submissions to W3C like [42] or [39]. Still, there is no official standard at the moment that focuses on semantics or behavioral specifications. Apart from the aforementioned possibilities, nothing prohibits from using any possible formal approach or any standard that is mainly proposed for WS* services but can be adapted to the specificities of RESTful services.

Service quality

WS*: Service quality generally concerns non-functional properties of the service, such as reliability, performance, security, privacy, trust, which characterize the quality of the results that the service promises to provide to its environment. Depending on the intended use of the service, non-functional properties are often considered less important than functional ones and their specification is omitted in service descriptions. Moreover, in contrast to functional service features, there is less agreement within the WS* community regarding the ways in which non-functional properties should be identified and specified.

Still, numerous efforts have produced languages and models for general or domain-specific quality description. Some of these efforts incorporate the expressiveness and reasoning power of ontologies. The Web Service Quality Model (WSQM) [55] is an ongoing standardization effort by OASIS for the specification of services quality. WSQM is a conceptual model; it defines a well-founded taxonomy of qualities and provides a wide range of non-functional properties. WSQM is layered, consisting of 3 main layers. The first layer, considers quality from a business perspective and is called Business Value Quality. The second layer, focuses on measurable qualities as perceived by the users while using services and is called Service-Level Measurable Quality. This quality defines properties like performance and stability. Then each property is associated with corresponding metrics such as response time, throughput, availability, successibility and accessibility. The third layer focuses on interoperability and management issues. The interoperability sub-layer concentrates on whether services can properly interoperate. The management sub-layer concerns whether the services provide external means for managing them. Then, accompanying the model, WS-QDL is a XML-based description language for representing non-functional properties of services by applying WSQM. Support for specifying non-functional properties is also considered in USDL [89], via modeling elements that shall be provided as parts of the pricing, the legal and the SLA modules.

RESTful: As in the case of WS* services, for RESTful services, there is no particular standard that focuses on the specification of non-functional properties.

2.1.2. Thing-based services

According to the Internet of Things overall view [90], in the near future it is expected that an ultra large number of devices will encompass computing and communication capabilities that will allow them to interact with their surrounding environment (including the physical world) and the inverse. Service-orientation in general is seen as a promising approach towards enabling the aforementioned collaboration. In this context, there have been interesting discussions that aimed at determining the prominent features of Thing-based services.

The two main service-oriented paradigms that were discussed in the previous sub-section offer two different alternatives for the realization of Thing-based services. Specifically, RESTful services are considered as the preferable approach for the realization of Thing-based services in the cases where the functionalities offered by Things are quite simple and atomic, in the sense that there are no complex conversations involved between Things and their surrounding environment [34].

On the other hand, for Things that offer more complex functionalities, WS* services are considered as a better option. Nevertheless, Things in general have limited computational/communication capabilities and resources [34]. Consequently, it is expected that Thing-based WS* services would comply to a limited subset of WS* standards that can be supported by Things [40]. In particular, it is expected that the bindings to Thing-based services would not rely on SOAP so as to avoid the complex encoding of SOAP messages. As opposed to that, simple HTTP-based bindings are considered as more reasonable. In the same spirit, most likely the XML parsing facilities that will be used for validating and parsing messages will be light-weight allowing only the parsing of messages that are actually relevant to the services. Moreover, Thing-based services would most probably be supported by light-weight compression and decompression facilities to cope with the limited resources of Things [63].

In many cases Thing-based services shall be event-based. Essentially, this means that the services interfaces would consist of oneway operations and that the services behavior would comply to typical event-based communication protocols such as the one proposed in the WS-Eventing standard [87]. According to these protocols a service may play the role of a subscriber, in which case the service sends out messages that signify the interest of the service for a certain type of events. Moreover, a service may play the role of an event source, meaning that that it accepts subscription requests and sends out notification messages to services that act as subscribers. Given the limited resources of Thing-based services, the management of subscriptions may be delegated to conventional services that act as subscription managers. Interaction protocols associated with the IoT in particular and the FI in general are further discussed in the next chapter.

Thing-based services would most likely be capable to support decentralized service discovery protocols such as the one proposed in the WS-Discovery standard [86]. Typically this means that the services behavior would comprise actions like multicasting/broadcasting: ”hello” messages, whenever they join a specific environment, ”bye” messages, whenever they leave the environment, probe messages, whenever they need to discover another service that is available in the environment.

Thing-based service descriptions would be typically accompanied with additional constraints. The constraints may be related to the digital context within which they can be used (e.g., the kinds of user applications that can use them), or to the physical context, within which they can be used (e.g., the location of the users). Finally, as opposed to conventional services where the specification of certain non-functional properties that characterize them is not a first priority, such characterizations are considered very important in the case of Thing-based services [34].

2.1.3. Formal definition of CHOReOS (service-oriented) components

Based on the previous discussion, Table 2.1 provides an overview of general paradigm-independent definitions related to the CHOReOS service-oriented components [6], which span Business and Thing-based services. Following, these definitions are discussed in further detail.

$$Service\ type \quad : \quad s = (n, p, l, I) \tag{2.1}$$
$$Interface \quad : \quad i = (n, p, O, b, C) \tag{2.2}$$
$$Operation \quad : \quad op = (n, p, In, Out, pre, post) \tag{2.3}$$
$$Service\ instance \quad : \quad si = (n, i, uri, d, nf, R) \tag{2.4}$$

Table 2.1: Definitions & notations related to CHOReOS components

Definition 1 (Service type) *Let Ω denote an infinitely countable set of domains and Σ denote an infinitely countable set of names. Then, a service type $s = (n, p, l, I)$ is defined as a tuple that consists of:*

- *A name, $s.n \in \Sigma$, that characterizes the service type.*
- *An optional service profile, $s.p$, such that $dom(s.p) \in \Omega$, that corresponds to a user-intuitive explanation of s; depending on the standard used this can vary between a simple textual description, a list of keywords, a more advanced description (e.g., DAML/OWL-S/SAWSDL) or any other description.*
- *A style, $s.l$ such that $dom(s.l) = \{WS^*, RESTful\} \in \Omega$, which refers to whether s conforms to the WS* or to the RESTful paradigm[2].*
- *A set of interfaces, $s.I$, that specify the functionalities provided by s.*

Definition 2 (Service interface) *A service interface $i = (n, p, O, b, C)$ is defined as a tuple that comprises:*

- *A name, $i.n \in \Sigma$, which characterizes the interface.*
- *An optional profile, $i.p$, such that $dom(i.p) \in \Omega$, that corresponds to a user-intuitive explanation of the interface.*
- *A set of operations, $i.O = \{op_1, \ldots, op_{|i.O|}\}$, that correspond to different functionalities provided through the interface. Note that this set may contain overloaded operations, i.e., operations with the same name and different input and output parameters.*
- *An optional behavioral specification, $i.b$, which, independently of specific standards (e.g., BPEL, WSCL) can be seen abstractly as a labeled transition system (LTS), i.e., $i.b : S \times L \rightarrow S$, where $S \in \Omega$ denotes a domain of states and $L \subseteq \Sigma$ denotes a set of labels.*
- *An optional set of constraints, $i.C = \{c_1, \ldots, c_{|i.C|}\}$, related to requirements over the environment where the interface functionalities execute. In general, a constraint $c_i \in i.C$ can be seen as a predicate defined over the interface's constituent elements and a domain of environmental properties $E \in \Omega$.*

Definition 3 (Interface operation) *An interface operation $op = (n, p, In, Out, pre, post)$ is a tuple that consists of:*

- *A name, $op.n \in \Sigma$, for the operation.*
- *An optional profile, $op.p$, such that $dom(op.p) \in \Omega$ that contains a user-intuitive explanation of the operation.*
- *A set of input parameters, $op.In = \{p_1, \ldots, p_{|op.In|}\}$ and a set of output parameters $op.In = \{p_1, \ldots, p_{|op.Out|}\}$. The set of output parameters may consist of a subset, $op.N \subseteq op.Out$, that comprises output parameters produced during the normal execution of the operation and a subset, $op.Ex \subseteq op.Out$, that comprises output parameters produced in exceptional situations. A parameter $p_i \in op.In \cup op.Out$ is generally defined as a tuple, $p_i = (n, p, t)$, that consists of a name, $p_i.n \in \Sigma$, an optional profile, $p_i.p$ such that $dom(p_i.p) \in \Omega$, and a type/domain $p_i.t \in \Omega$.*
- *An optional pre-condition, $op.pre$, that must hold for the correct execution of the operation. The pre-condition is generally defined as a predicate over the operation's constituent elements.*

[2] As already mentioned and further detailed in the next chapter, core Web-based service paradigms are called to evolve to face the FI challenges, in particular regarding the interaction paradigms that need to be supported. However, the impact of such an evolution upon the one of components is not documented in this deliverable and will be detailed in the next one, i.e., D1.4.

- *An optional post-condition, $op.post$, that shall hold after the correct execution of the operation. As in the case of the pre-condition, the post-condition is generally defined as a predicate over the operation's constituent elements.*

Following the definition of CHOReOS components we provide a definition of CHOReOS component instances.

Definition 4 (Service/Component instance) *Let $s = (n, p, s, I)$ be a type of services that follows the previous definitions. Then, s defines a domain of service instances (hereafter we use the term service to refer to an instance of a service type), where a particular service instance $si = (n, i, uri, d, nf, R)$ is defined as a tuple that consists of:*

- *A service name, $si.n \in \Sigma$.*
- *The interface $si.i \in s.I$ of the service type s that is implemented by si.*
- *An endpoint address, $si.uri$ such that $dom(si.uri) \in \Omega$.*
- *An optional description of implementation related details (e.g., the protocol used for accessing the service functionalities, the message format, etc.), $si.d$, such that $dom(si.d) \in \Omega$.*
- *An optional description of non-functional properties that characterize si. Specifically, the non-functional description $si.nf = [qp_1, \ldots, qp_{|nf|}]$ is a vector of quality properties, where each property $si.nf[qp_i] = (q_i, value)$ is a tuple that consists of a quality indicator $q_i \in Q$ that belongs in a domain of quality indicators $Q \in \Omega$ and a $value \in dom(q_i)$ that belongs to the corresponding quality indicator domain. In general, we do not restrict the non-functional characterization of services to a standard domain of quality indicators; on the contrary, we employ this formation to allow flexibility on this kind of descriptions. Nevertheless, we can distinguish between runtime quality indicators such as reliability, availability, reputation, price, performance [1], design quality indicators such as different kinds of cohesion for service interfaces discussed in [4] and finally physical properties that typically characterize Thing-based services.*
- *An optional set of requirements $si.R = \{rs_1, \ldots, rs_{|s.R|}\}$ for services that may be required by a si in the case where si is a composite service. Each requirement $rs_i = (r_s, r_{nf}) \in si.R$ is characterized by the type of the required service $rs_i.r_s$ and a vector of required non-functional properties, $rs_i.r_{nf}$.*

Taking a concrete example, in the *RemoteMethods*[3] registry we find various services that allow sending SMS messages to mobile phones. `SMS-TXT`[4] is such a service that follows the WS* paradigm. In the WSDL specification of the service interface we have a single operation named `SendSms`. The operation accepts as input message, five string parameters. The output message of the operation is empty. With respect to the CHOReOS components definition, the service interface are specified as showed in the tabular representation given in Figure 2.1(a). The interface does not include profile specifications since there is no such information available in the registry where the service was found. For the same reason, there are no behavioral specifications and constraints.

An example of the CHOReOS representation of a RESTful service interface is given in Figure 2.1(b). The example is based on the Yahoo news search application [88]. This application consists of a single resource that provides a single method called `search`. Consequently in the CHOReOS representation, we have an interface that defines a corresponding operation.

Finally, another example of an SMS service interface is given in Figure 2.1(c). In particular, `GlobalSMSPro`[5] also follows the WS* paradigm. The service offers a more complex interface. As in

[3] `http://www.remotemethods.com/`.

[4] `http://www.sms-txt.co.uk/sendSms.asmx`.

[5] `http://www.strikeiron.com/Apps/runapp.aspx?appid=95`.

Interface SendSmsHttpPost			
SendSmsHttpPost.**O**			
SendSms			
In	**p.n**	**p.t**	**p.p**
	FromName	string	empty
	FromNumber	string	empty
	ToNumber	string	empty
	Message	string	empty
	Locale	string	empty
Out	**p.n**	**p.t**	**p.p**
	empty	empty	empty
p	empty		
pre	empty		
post	empty		
b	empty		
C	empty		

(a) `SMS-TXT` service.

Interface newsSearch			
newsSearch.**O**			
search			
In	**p.n**	**p.t**	**p.p**
	appid	string	empty
	query	string	empty
	type	string	empty
	results	int	empty
	start	int	empty
Out	**p.n**	**p.t**	**p.p**
	totalResultsAvailable	int	empty
	totalResultsReturned	int	empty
	summary	string	empty
	……		
p	empty		
pre	empty		
post	empty		
b	empty		
C	empty		

(b) Yahoo `newsSearch` service.

Interface SMSTextMessagingSoap							
SMSTextMessagingSoap.**O**							
SendMessage				SendMessageBulk			
In	**p.n**	**p.t**	**p.p**	**in**	**p.n**	**p.t**	**p.p**
	ToNumber	string	empty		ToNumbers	string[*]	empty
	FromNumber	string	empty		FromNumber	string	empty
	FromName	string	empty		FromName	string	empty
	MessageText	string	empty		MessageText	string	empty
Out	**p.n**	**p.t**	**p.p**	**out**	**p.n**	**p.t**	**p.p**
	ToNumber	string	empty		ToNumber	string[*]	empty
	TrackingTag	string	empty		TrackingTag	string[*]	empty
	StatusCode	int	empty		StatusCode	int[*]	empty
	StatusText	string	empty		StatusText	string[*]	empty
	StatusExtra	string	empty		StatusExtra	string[*]	empty
p	empty			**p**	empty		
pre	empty			**pre**	empty		
post	empty			**post**	empty		
b	empty						
C	empty						

(c) `GlobalSMSPro` service.

Figure 2.1: Examples of CHOReOS component related specifications

the previous case, the interface does not include information regarding semantics, behavior, constraints and non-functional properties, since there is no such information available in the registry where the service was found. The interface consists of 6 operations. The `SendMessage` and `SendMessagesBulk` operations that are given in Figure 2.1(c) are the ones that actually send SMS messages to mobile phones, while the rest of the operations (omitted in Figure 2.1(c) for reasons of simplicity) serve for monitoring the status of SMS messages, or finding information concerning different country codes, network standards, etc.

2.2. Formal Abstraction for FI Services

In the early 70's, Parnas [59] introduced the fundamental principle of information hiding and discussed the benefits of abstractions, which hide the details of various alternative design options, for developing software. Today, all over the Web, we have a plenitude of alternative design options, provided in the form of reusable services. The amount of these options is constantly growing. Given the results reported in Deliverable D1.2 [77], the annual growth rate of the number of Business services that become available

through the Web is 130%. The number of Thing-based services is anticipated to be even larger in the FI. Besides the availability of a plenitude of services, today, we further have discovery protocols, crawlers and Web service search engines [26, 1] that allow the discovery of these large amounts of available services. Overall, this growing amount of alternative design options points out that in the Future Internet it will be more than ever necessary to employ higher-level abstractions.

Service discovery, composition and adaptation can benefit from the concept of service abstractions that represent semantically compatible services (i.e., services which provide the same functionality, possibly, through different interfaces). As previously mentioned, various semantic description languages have been proposed for the specification of service abstractions (e.g., OWL-S). Moreover, certain service registries (e.g., [27, 65, 57]) already assume certain notions of abstraction, towards the organization of information concerning available services. Finally, several approaches for service composition assume the existence of service abstractions (e.g., [92, 93, 23, 3, 19, 20, 5]). Then, the proposed approaches provide means for adapting compositions of service abstractions to meet changes in functional and non-functional requirements. The adaptation takes place, by substituting the concrete services that are hidden behind the composed service abstractions.

The CHOReOS style goes beyond these early approaches by providing formally grounded service abstractions that can be hierarchically structured. Moreover, the CHOReOS style distinguishes between two different types of interrelated service abstractions: *functional abstractions that represent groups of services that offer similar functional properties* and *non-functional abstractions that represent groups of services that provide similar non-functional properties*.

The CHOReOS abstractions are considered as core elements that are reverse engineered from the services that become available over time and assist the task of service discovery and the subsequent tasks of service composition and adaptation. To assist the discovery of a service that provides specific functional/non-functional properties, we assume as input the developers/domain-experts' functional/non-functional requirements, as well as one or more already produced hierarchies of functional abstractions. Then, the discovery process amounts to:

- Finding a functional abstraction that satisfies any or most of the functional requirements.
- Returning the matching functional abstraction along with the services that are described by it.

The discovery process may return too many functionally equivalent services, thus complicating the selection of the appropriate service (we recall here that the ultra large scale of the Future Internet raises these challenges). This issue becomes even more important considering that the actual service selection may take place at runtime. Hence, we need to provide extra selection criteria; we do this by utilizing the non-functional properties of services (e.g., response time, availability, throughput etc.). So, the selection of an actual service from the set of services elicited in the previous step consists of:

- Browsing or searching for the suitable non-functional abstraction that meets certain non-functional requirements.
- Selecting any of the services of the selected non-functional abstraction.

In the rest of this section, we formally define the abstraction driven organization of available services assumed by the CHOReOS architectural style. Based on the proposed organization, we further define formally basic primitives for abstraction-driven service registration and discovery, which are exploited within WP2 and WP3 for the respective development (from design to prototype implementation) of the CHOReOS abstraction-oriented service base and eXtensible discovery service.

2.2.1. Abstraction-driven organization of services

In the FI, information about available services may be obtained from various sources. A source may be a service provider, a service registry/portal, a distributed service discovery protocol, etc. Then,

in the CHOReOS architectural style, we assume that information about services is organized in the CHOReOS abstraction-oriented service base, which is formally defined as follows.

Definition 5 (Abstraction-oriented service base) *The CHOReOS abstraction-oriented service base* $sb = (C, FA, NFA)$ *is defined as a tuple that consists of:*

- *A set of service collections* $sb.C = \{c_1, \ldots, c_{|sb.C|}\}$*. Each collection* $c_i \in sb.C$ *is a set of services. In general, we can distinguish 2 different kinds of service collections, those that directly reflect the information about existing services that is collected by a single source, and those that contain services information that comes from multiple sources. The latter collections may result from the union of other collections, from the results of user-defined service discovery queries, etc.*
- *A set of functional abstractions hierarchies* $sb.FA = \{fa_1, \ldots, fa_{|sb.FA|}\}$*. Specifically, each element* $fa \in sb.FA$ *is a hierarchically structured functional abstraction that is reverse engineered from a particular collection of services* $c \in sb.C$*.*
- *A set of non-functional abstractions hierarchies* $sb.NFA = \{nfa_1, \ldots, nfa_{|sb.NFA|}\}$*. Specifically, each element* $nfa \in sb.NFA$ *is a hierarchically structured non-functional abstraction that is reverse engineered from a particular collection of services* $c \in sb.C$*, or from the services that are represented by a specific functional abstraction* $fa \in sb.FA$*.*

Functional abstractions

A functional abstraction serves the purpose of acting as the representative of a set of services. In a sense, a functional abstraction can be seen as an abstract type/domain, while the types/domains of the represented services correspond to subtypes/subdomains of this abstract type/domain. In this context, Liskov & Wing defined in [44] a list of fundamental rules (invariants, history, pre-conditions, post-conditions, contra-variance, and co-variance), which guarantee that a type S is a correct subtype of a type T. To define the notion of functional abstraction we considered these fundamental rules. However, some of these criteria cannot be directly applied in the case of functional abstractions.

Specifically, the invariants and history rules are state-related constraints, which are not applicable to services since their descriptions do not reveal state information. Concerning the pre-conditions and post-conditions rules [44], certain service description languages provide means for specifying pre-conditions and post-conditions. However, our experience with several collections of available services, shows that pre-conditions and post-conditions are rarely provided. Nevertheless, in the definition of functional abstractions these aspects are taken into account along with the typical contra-variance and co-variance rules. Briefly, the contra-variance rule states that every method m_S of the type S is mapped to a method m_T of the T, such that the type of each argument of m_T is a subtype of the type of the corresponding argument of m_S. Moreover, the pre-condition rule states that the pre-condition of m_T implies the pre condition of m_S. The co-variance rule states that the type of the result of m_S is a subtype of the result of m_T. The post-condition rule requires that the post-condition of m_S implies the post-condition of m_T. In the context of SOA, the methods correspond to interface operations, whose arguments and results correspond to input and output messages.

Then, an overview of the definition of functional abstractions is given in Table 2.2. Following, this definition is discussed in further detail.

Definition 6 (Functional abstraction) *Given a service base* sb*, we define a functional abstraction* $fa = (i, R, M, anc, desc) \in sb.FA$ *that is reverse engineered from a collection of services* $c \in sb.C$ *as a tuple that comprises the following elements:*

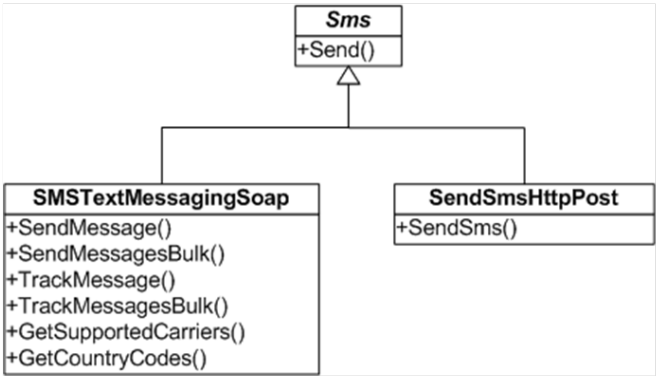
- *A set of services* $fa.R = \{si_1, \ldots, si_{|fa.R|}\}$*, which are represented by* fa*. This set of services may be the collection* c *it self or a subset of* c*, i.e.,* $fa.R \subseteq c$*.*

$$Functional\ abstraction \ : \ fa = (i, R, M, anc, desc) \quad (2.5)$$
$$Mapping\ between\ fa.I\ and\ r_i \in fa.R \ : \ m_{ri_i} = (m_{op}, M_{In}, M_{Out}) \quad (2.6)$$
$$Operation\ mapping \ : \ m_{ri_i}.m_{op} : fa.i.O \rightarrow ri_i.O \quad (2.7)$$
$$Inputs\ mapping\ between\ op \in fa.I.O\ and\ m_{op}(op) \ : \ m_{in} : op.In \rightarrow m_{ri_i}.m_{op}(op).In \quad (2.8)$$
$$Outputs\ mapping\ between\ op \in fa.I.O\ and\ m_{op}(op) \ : \ m_{out} : op.Out \rightarrow m_{ri_i}.m_{op}(op).Out \quad (2.9)$$
$$Non-functional\ abstraction \ : \ nfa = (nf, R, anc, desc) \quad (2.10)$$

Table 2.2: Definitions & notations related to the CHOReOS abstractions

- *An abstract interface, $fa.i$, whose operations correspond to common/similar operations offered by the interfaces of the represented services $fa.R$.*
- *A set of mappings, $fa.M = \{m_{s_1}, \ldots, m_{s_{|fa.R|}}\}$, between the abstract interface $fa.i$ and the interfaces of the represented services. Specifically a mapping $m_{s_i} = (m_{op}, M_{In}, M_{Out}) \in fa.M$ between $fa.i$ and the interface ri_i of a represented service s_i is defined as a tuple that consists of:*
  - *A function $m_{ri_i}.m_{op} : fa.i.O \rightarrow ri_i.O$ between the operations $fa.i.O$ of the abstract interface and the operations $ri_i.O$ of the represented interface.*
    With respect to the Liskov & Wing pre/post condition rules, we assume that the mapping $m_{ri_i}.m_{op}$ is strictly well-formed if the following conditions hold for each pair of mapped operations $op \in fa.i.O$ and $m_{ri_i}.m_{op}(op) \in ri_i.O$:
    1) $op.pre \Rightarrow m_{ri_i}.m_{op}(op).pre$
    2) $m_{ri_i}.m_{op}(op).post \Rightarrow op.post$
  - *A set of mappings $m_{ri_i}.M_{In}$ between the input parameters of mapped operations and a set of mappings $m_{ri_i}.M_{Out}$ between the output parameters of mapped operations. In particular, given the mapping $m_{ri_i}.m_{op} : fa.i.O \rightarrow ri_i.O$ between the operations $fa.i.O$ of the abstract interface and the operations $ri_i.O$ of the represented interface ri_i, for each pair of mapped operations $op \in fa.i.O$ and $m_{ri_i}.m_{op}(op) \in ri_i.O$ we have:*
    - *$m_{ri_i}.M_{In}$ contains a function $m_{in} : op.In \rightarrow m_{ri_i}.m_{op}(op).In$ for the inputs of the mapped operations.*
      With respect to the Liskov & Wing contra-variance rule, we assume that m_{in} is well-formed if for every pair of mapped input parameters $p \in op.In$ and $m_{in}(p)$ the type of p is a sub-type of the type of $m_{in}(p)$.
    - *$m_{ri_i}.M_{Out}$ contains a function $m_{out} : op.Out \rightarrow m_{ri_i}.m_{op}(op).Out$ for the outputs of the mapped operations.*
      With respect to the Liskov & Wing co-variance rule, we assume that m_{out} is well-formed if for every pair of mapped output parameters $p \in op.Out$ and $m_{out}(p)$ the type of $m_{out}(p)$ is a subtype of the type of p.
  - *In general, certain subsets of the services that are represented by fa may be further organized with respect to lower-level functional abstractions. Hence, fa is further characterized by a set of such lower-level abstractions, $fa.desc$. Moreover, the services that are represented by fa may be a subset of services organized with respect to a higher-level functional abstraction $fa.anc$.*

Getting back to the example of the two different SMS services that was previously discussed (Figure 2.1), an interesting observation is that although these two services come from two different service



Abstract Interface Sms			
Sms.**O**			
Send			
In	**p.n**	**p.t**	**p.p**
	FromName	string	empty
	FromNumber	string	empty
	ToNumber	string	empty
	Message	string	empty
Out	**p.n**	**p.t**	**p.p**
	empty	empty	empty
p	empty		
pre	empty		
post	empty		
b	empty		
C	empty		

Figure 2.2: Example of a service functional abstraction

providers their interfaces offer at least a pair of very similar operations. Specifically, the `SendSms` operation of the `SMS-TXT` service interface and the `SendMessage` operation of the `GlobalSMSPro` interface realize semantically compatible functionalities (i.e., sending an SMS message) and further have very similar input parameters. This observation points out an opportunity for defining a service abstraction that represents these two services. A possible interface for this abstraction is given in Figure 2.2. In particular, the abstract service interface that represents the interfaces of the two services comprises a single operation named `Send` and requires 4 string parameters. The mapping of this abstract interface to the interfaces of the two services is rather straightforward and is given in Figure 2.3(a). Finally, Figure 2.3(b) gives an example of a mapping between the input parameters of the abstract operation `Send` and the input parameters of the `SendSms` operation of the `SMS-TXT` service interface.

Although the definition of a functional abstraction seems trivial for the two services of our example, in the general case, where the given set of available services is going to be large and the services much more complex, this task is not going to be easy. Consequently, in CHOReOS, as part of WP2 work we develop methods for reverse-engineering functional abstractions out of a given set of available services. Further details concerning the reverse engineering of functional abstractions are not in the scope of this document. However, we provide here a general definition.

Definition 7 (Functional abstraction recovery) *A method for the recovery of functional abstractions is defined as an algorithm that accepts as input a collection of services* $c \in sb.C$ *that belong to the service base* sb *and computes a hierarchy of functional abstractions* fa*, which is then included in* $sb.FA$*, i.e.,* $alg_{fn} : sb.C \rightarrow sb.FA$*.*

Non-functional abstractions

The services of a particular collection $c \in sb.C$ that belongs to the abstraction-oriented service base sb, or the services $fa.R$ that are represented by a functional abstraction $fa \in sb.FA$ may be further organized into groups of services that offer similar non-functional properties. These groups are represented by a corresponding non-functional abstraction $nfa \in sb.NFA$. Hence, a non-functional abstraction could be simply defined as a group of services, which is further characterized by a vector of quality properties, as it is done in the case of individual services. The value of each non-functional property could be, for instance, the average/min/max of the values that characterize the quality property of the represented services.

Taking the example of the SMS services, suppose our quality indicators are reputation, price and availability. Moreover, assume that `SMS-TXT`, `GlobalSMSPro` and a third service

operations mapping between
the abstract interface and the GlobalSMSPro interface

	Sms	
		Send
SMSTextMessagingSoap	**SendMessage**	X
	SendMessageBulk	
	TrackMessage	
	TrackMessageBulk	
	GetCountryCodes	
	GetSupportedCarriers	

operations mapping between
the abstract interface and the SMS-TXT interface

	Sms	
		Send
SendSmsHttpPost	**SendSms**	X

(b) Operation mappings.

		Send			
		ToNumber string	**FromNumber string**	**FromName string**	**Message string**
SendSms	**FromName string**			X	
	FromNumber string		X		
	ToNumber string	X			
	Message string				X
	locale string				

(b) Parameter mappings.

Figure 2.3: Example of a service functional abstraction - operation and parameter mappings

`SMS-Text-Messaging`[6] are characterized by corresponding quality properties, the values of which are given in Figure 2.4. Then, a non-functional abstraction that represents these three services could be characterized by the average value of reputation, the maximum price, and the minimum availability (i.e., $[reputation = -0.1, price = 55EURO, availability = 0,8]$). However, often the developers/domain-experts are not interested in concrete values when browsing groups of services that have similar non-functional properties; instead there are interested in more abstract quality characterizations (e.g., $[reputation = neutral, price = cheap, availability = acceptable]$). For that reason in the case of non-functional abstractions we introduce the concept of *domain hierarchies* that characterize quality indicators which is defined as follows.

Definition 8 (Domain hierarchy) *Let* $\mathbf{Q}$ *denote a countable set of quality indicators of interest and* $\mathbf{\Omega}$ *denote an infinitely countable set of domains. Each quality indicator* $q \in Q$ *is characterized by a concrete domain* $dom(q) \in \Omega$*. The domain can be a discrete or dense set of acceptable values for the quality indicator (e.g., the quality indicator reliability can have values in the dense domain of [0..1]). Then, a domain hierarchy* $\mathbf{H} = \{\delta_1, \ldots, \delta_{|H|}\}$ *for* q *is a finite* list *of domains* $\{\delta_1, \ldots, \delta_n\}$ *such that* $\delta_1 = dom(q)$*. The following constraints are further required for these domains:*

[6] http://ws.strikeiron.com/globalsmspro2_5?WSDL.

	reputation	*price*	*availability*
SMS-TXT	-0.7	50 EURO	0.8
GlobalSMSPro	-0.1	60 EURO	0.95
SMS-Text-Messaging	0	55 EURO	0.85

Figure 2.4: A set of services with their non-functional properties

- *For every pair of domains* $\delta_{low}, \delta_{high}$ *such that* $low < high$ *(i.e.,* δ_{low} *is found lower in the hierarchy than* δ_{high}*) a total, onto* ancestor *function is defined for their members* $\alpha_{\delta_{low}}^{\delta_{high}} : \delta_{low} \rightarrow \delta_{high}$*.*
- $\forall i < j < k, x \in \delta_i, y \in \delta_j, z \in \delta_k | \alpha_{\delta_i}^{\delta_j}(x) = y \wedge \alpha_{\delta_j}^{\delta_k}(y) = z \Rightarrow \alpha_{\delta_i}^{\delta_k}(x) = z$
- *For every* $\delta_{low}, \delta_{high} \in H$*, if* δ_{low} *is ordered then,* $\forall x, y \in \delta_{low} | x < y \Rightarrow \alpha_{\delta_{low}}^{\delta_{high}}(x) \leq \alpha_{\delta_{low}}^{\delta_{high}}(y)$*.*

We say that a domain $delta_{high}$ *is* more general*, or* subsumes*, or* dominates *another domain* $delta_{low}$ *in the same hierarchy whenever* $low < high$ *and the function* $\alpha_{\delta_{low}}^{\delta_{high}}$ *is defined.*

Intuitively, a domain hierarchy is a list of abstraction levels, each providing more abstract characterizations of the values of the domains found lower in the lower in the hierarchy. The lower a position in the hierarchy's list, the more detailed the domain is. The values of the domains in the hierarchy are mapped to each other via an ancestor function α. We require that α is (a) a function, (b) total and (c) onto. In other words, all the values of a lower domain are mapped to a higher level and the values of higher levels in the hierarchy always have descendants. Also, we require that there is a consistency constraint between the members of the domains. Specifically, since one can ascend from level say δ_2 to level δ_5 either directly, via the function $\alpha_{\delta_2}^{\delta_5}$ or via the composition of different functions over the intermediate levels, we require that the result of all these paths over the values of the involved domains result in the same value in δ_5. By definition, a more general domain abstraction has at the most the same cardinality with a detailed one. Note also that the inverse of α is not a function. Finally, well-formed hierarchies map continuous ranges of detailed values to abstract values. Equivalently, a well-formed abstraction imposes an equivalence relation that partitions the detailed domain in continuous ranges of values.

In Figure 2.5, we depict examples of domain hierarchies for reputation, price and availability. The dotted edges signify the ranges of the ancestor functions between two levels. Observe that the lowest level is dense whereas the others are discrete, without any theoretical problems. Also, observe that the ancestor function involves ranges which is a typically expected case for domains where an ordering can be defined with an intuitive manner, but not obligatory.

Then, based on the above we define non-functional abstractions as follows.

Definition 9 (Non-functional abstraction) *Given a set of quality indicators* $Q = \{q_1, \ldots, q_{|Q|}\}$ *(each quality indicator* $q_i \in Q$ *is characterized by a concrete domain* $dom(q_i) \in \Omega$*) and a set of domain hierarchies* $S_H = \{H_1, \ldots H_{|S_H|}\}$ *defined for the set of quality indicators* Q*, a non-functional abstraction* $nfa \in sb.NFA$ *is defined as a tuple* $nfa = (nf, R, anc, desc)$ *where:*

- $nfa.R$ *is the set of the services represented by* nfa*.*
- $nfa.nf = [qp_1, \ldots, qp_{|nf|}]$ *is a vector of quality properties, where each property* $nfa.nf[qp_i] = (q_i, value)$ *is a tuple that consists of a quality indicator* $q_i \in Q$ *that belongs in* Q *and a* $value \in \delta_{q_i}$*,*

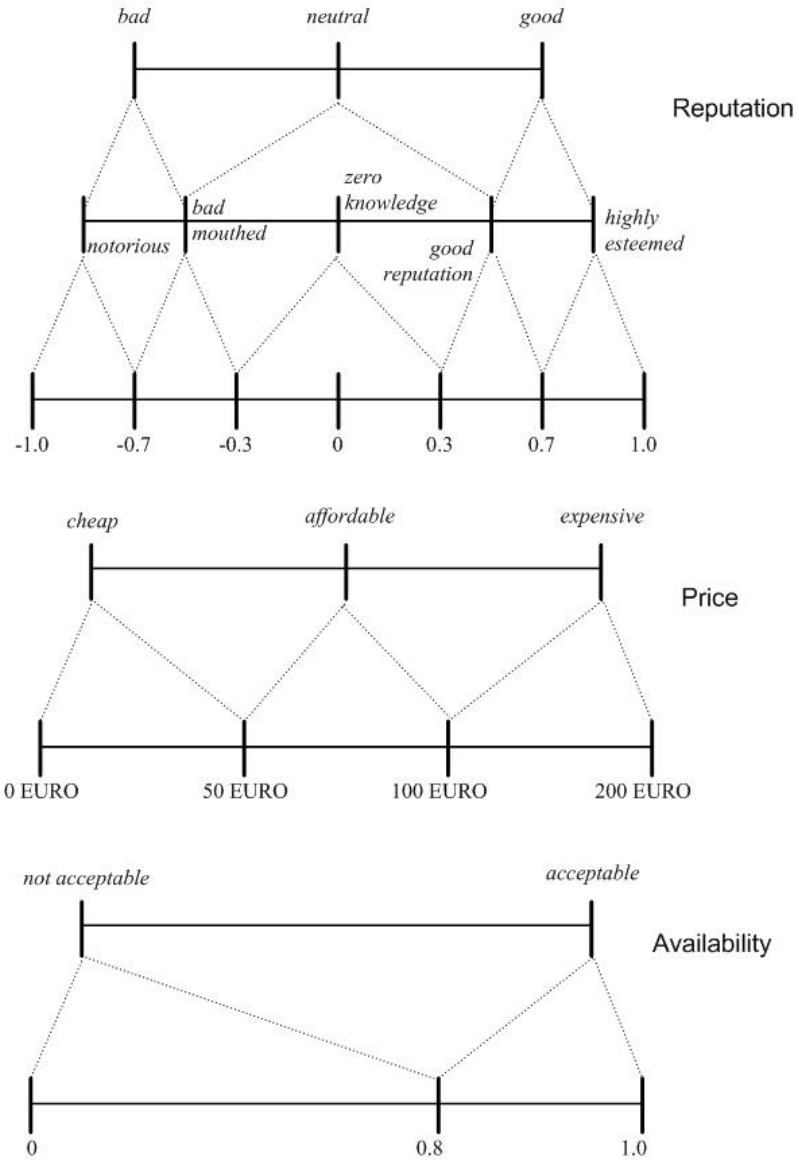


Figure 2.5: Domain hierarchies for reputation, price and availability

the domain of which belongs to the domain hierarchy $H_i \in S_H$ that corresponds to the quality indicator q_i.

- *For each service $s \in nfa.R$ and for each quality indicator $q_i \in Q$, $nfa.nf[qp_i].value = \alpha^{\delta_{q_i}}_{dom(q_i)}(s.nf[qp_i].value)$. In other words, all the services of the non-functional abstraction are characterized by the same abstract values for all their quality indicators.*
- *Certain subsets of the services that are represented by nfa may be further organized with respect to lower-level non-functional abstractions. Hence, nfa is further characterized by a set of such lower-level abstractions, $nfa.desc$. Moreover, the services that are represented by nfa may be a subset of services organized with respect to a higher-level functional abstraction $nfa.anc$.*

Going back to our example, based on the non-functional properties of the services of Figure 2.4 and the domain hierarchies of Figure 2.5 it is possible to define a non-functional abstraction that groups the three services with the characterization $[reputation = neutral, price = affordable, availability = acceptable]$. Moreover, it is possible to define two different abstractions, one for representing `GlobalSMSPro` and `SMS-Text-Messaging` with the characterization

$[reputation = zero-knowledge, price = affordable, availability = acceptable$ and the second one that represents only SMS-TXT with the characterization $[reputation = bad-mouthed, price = affordable, availability = acceptable$

Similarly, to the case of functional abstractions, the organization of services with respect to non-functional abstractions requires corresponding methods for reverse engineering such abstractions out of existing services. A general definition of the reverse engineering of non-functional abstractions is provided below. However, these methods are developed as part of WP2 and generally. Hence, the exact mechanics of how the non-functional abstractions are obtained are not in the scope of this document.

Definition 10 (Non-functional abstraction recovery) *A method for the recovery of non-functional abstractions is an algorithm that accepts as input either a collection of services $c \in sb.C$ that belong to the abstraction-oriented service base sb or a set of services that are represented by a functional abstraction $nfa \in sb.FA$ and computes a hierarchy of non-functional abstractions nfa, which is included in $sb.NFA$, i.e., $alg_{nfn} : sb.C \bigcup_{\forall fa \in sb.FA}(\{fa.R\}) \rightarrow sb.NFA$.*

2.2.2. Abstraction-driven registration and lookup of services

Managing the abstraction-oriented service base $sb = (C, FA, NFA)$ that is organized with respect to functional and non-functional abstractions amounts to further providing means that allow to register information about existing services in this organization. The registration of services information concerns either the creation of a new collection of services, or the upgrade of an existing collection of services. Although the details of these tasks concern corresponding methods developed in the context of WP2, we provide here the following general definitions.

Definition 11 (Service registration) *A method for the registration of services information in the abstraction-oriented service base $sb = (C, FA, NFA)$ is an algorithm $alg_{reg} : Src \times P(sb.C) \rightarrow sb.C$ that may accept as input a source selected from a set of known sources Src that provide information about services. Alternatively, the algorithm may accept as input a set of existing collections of services. Then, the algorithm constructs a new collection of services by retrieving information about available services from the source, or merging the contents of the given collections.*

Definition 12 (Refreshment) *A method for the refreshment of a collection that already exists in the abstraction-oriented service base $sb = (C, FA, NFA)$ is an algorithm $alg_{rfs} : Src \times sb.C \rightarrow sb.C$ that accepts as input the collection and the source used for producing the collection. Then, the algorithm upgrades the contents of the existing collection. This task may further trigger the upgrade of related abstractions hierarchies. If the abstractions hierarchies radically change (i.e., new abstractions are introduced in the hierarchies) the previous versions may be kept either for a limited time or for as long as they are needed (i.e., there exist choreographies that rely on these hierarchies).*

Exploiting the contents of the abstraction-oriented service base involves browsing the functional and non-functional abstractions hierarchies, or posing queries. Again, the details of the querying method concern WP2. However we provide here the following general definition.

Definition 13 (Service lookup) *Executing a query over the abstraction-oriented service base $sb = (C, FA, NFA)$ involves an algorithm $alg_q : dom_q \rightarrow \bigcup_{\forall c_i \in sb.C}(P(c_i)) \times P(sb.FA) \times P(sb.NFA)$ that accepts as input a query expression $q \in dom_q$, such that $dom_q \in \Omega$. A query expression to the abstraction-oriented service base is a tuple q=(c_i, ϕ_{ST},ϕ_S,γ, λ), where:*

- *$c_i \in sb.C$ is a collection of the abstraction-oriented service base.*
- *ϕ_{ST} is an (optional) expression over the properties of the service type and interface of the services of the collection c_i.*

- *ϕ_S is an (optional) expression over the properties of the services of the collection c_i.*
- *γ is an (optional) expression involving a list of functional or non-functional abstractions, annotated with an execution tag that restructures the result in groups.*
- *λ is a (optional) list of features (of the service type/interface or the service) that we want to see exported as part of the answer.*

We assume that ϕ_{ST} and ϕ_{ST} are expressed as conjunctions of atomic expressions of the form ($feature\ \theta\ value$), where $feature$ is a functional or non-functional property as specified before, $value$ is a value in the appropriate domain and θ belongs to the set $\{ =, \neq, <, >, \in, \supseteq, \subseteq, \ldots, \}$.

The grouper list γ can at most involve (a) an expression over the functional abstractions and (b) an expression over the non-functional abstractions of the collection. Both expressions can specify whether they want `existing` abstractions or `run-time` abstractions to be generated. In the latter case, concerning the non-functional abstractions, the user might specify a subset of the possible quality indicators for the formation of the abstractions.

The semantics of the query (i.e., how the result is computed given the query expression) is as follows:

1) *First, the services belonging to c_i are retrieved.*
2) *If present, the filters ϕ_{ST} and ϕ_{ST} are applied restricting the set of services to a result set R that includes only those services of c_i that satisfy them.*
3) *Then, the services of R are restructured according to the set of functional or non-functional abstractions that accompany the collection. The user might specify one of the following: (a) reuse the existing abstractions, or (b) compute new abstractions over the result set R. If the user specifies both functional and non-functional abstractions as groupers, the results nest non-functional groups under the functional ones.*
4) *Then, if specified, out of the possible attributes that can be related to the retrieved services, the ones belonging to λ are retained.*

Intuitively, the above definition refers to the following request: "Take a collection of services and return back a subset of this collection". The services belonging to the returned subset must fulfill the constraints specified by filters ϕ_{ST} and ϕ_S. For example, the user might ask for services whose interface involves an operation, whose name contains the term $Send$ and input signature consists of at least two $string$ parameters, whose names contain the terms $From$, To (see example in Figure 2.2). Moreover, in the previous query the user may ask for the services whose $reliability$ is above 0.75. Once these services are retrieved, the user might also specify that he would like an abstraction-driven organization of the services. For instance, an organization with respect to non-functional abstraction may be requested such that groups of services will be formed based on the $price$, so that all expensive services are found in one group, the cheap ones in another and so on and so forth. Finally, the user may also specify that out of all the characteristics of the retrieved services, he is interested in their interface's $operations$, their $endpoints$ and their $availability$ and $reputation$."

3 CHOReOS Connectors: Interoperability across Interaction Paradigms

With developments such as ubiquitous computing possibly coupled with enhanced modes of networking (e.g., using mobile ad hoc networks), mobile computing with an increasing range of always-connected, portable terminals allowing for novel distributed system services, and cloud computing enabling complex distributed system services to be offered in the greater Internet, the level of heterogeneity in distributed systems has increased dramatically in the recent years [13]. It is then inevitable that complex distributed applications in the Future Internet will be to a large extent based on the open integration of extremely heterogeneous systems, such as lightweight embedded systems (e.g., sensors, actuators and networks of them), mobile systems (e.g., smartphone applications), and resource-rich IT systems (e.g., systems hosted on enterprise servers and Cloud infrastructures). These heterogeneous system domains differ significantly in terms of interaction paradigms, communication protocols, and data representation models, which are most often provided by supporting middleware platforms. In particular with regard to middleware-supported interaction, the client/server (CS), publish/subscribe (PS), and tuple space (TS) paradigms are among the most widely employed ones today, with numerous related middleware platforms, such as: Web Services, Java RMI for CS; WS-Eventing[1], JMS, SIENA for PS [51, 21]; and JavaSpaces, Lime for TS [31, 52]. Those paradigms are further evolving towards enabling interaction in the Internet of Things, which bring any Thing in the network, and in particular Sensors and Actuators (S & A) to be able to interpret and act upon the physical world. State of the art evolution of the interaction paradigms for S & A is illustrated by latest research effort in the area of integration of WSAN (Wireless Sensor and Actuator Networks) with Business services (e.g., see [28, 7, 34, 8]). Integration ranges from the integration of an overall WSAN, a sub-WSAN to an individual Thing, as a service. Such evolution primarily builds upon the core interaction paradigms that are CS, PS and TS, further leading to systematize support for both discrete and continuous interaction under any of those paradigms.

In light of the above, the connector types associated with the CHOReOS architectural style shall (i) leverage the diversity of interaction paradigms associated with today's and future complex distributed systems, as well as (ii) enable cross-paradigm interaction to sustain interoperability in the highly heterogeneous Future Internet. The former aspect straightforwardly lays in the definition of the corresponding connector types. The latter is concerned with solving architectural mismatches arising between connected components, for which we aim at an automated solution in order to sustain cross-domain interoperability and further allow for truly open and dynamic system integration. As surveyed in [37, 75], existing cross-domain interoperability efforts are based on, e.g., bridging communication protocols [11], wrapping systems behind standard technology interfaces [8], and/or providing common API abstractions [33, 22, 91, 62]. In particular, such techniques have been applied by the two core system integration paradigms adopted by CHOReOS, that is, service oriented architecture (SOA) and enterprise service bus (ESB) [58]. Indeed, SOA enables representing and accessing systems as autonomous components *via* standard interfaces, while ESB provides a backbone infrastructure for SOA by enabling bridging among heterogeneous systems by means of a common middleware-level bus protocol. More precisely, a system is connected to the ESB *via* a middleware adapter, which adapts the middleware platform employed by the system to the common bus protocol, and exposes on the bus a SOA interface

[1] http://www.w3.org/TR/ws-eventing/

for the system.

However, state of the art interoperability efforts commonly cover part of the heterogeneity issues (regarding interaction, communication, data) and are applicable to specific cases [13]. In particular, existing solutions do not or only poorly address interaction paradigm interoperability. Specifically, SOA and ESB are primarily based on the CS paradigm. Even if extensions have been proposed, such as event-driven SOA or ESB supporting the PS paradigm [58], these remain partial. This means that systems integrated via SOA and ESB solutions have their interaction semantics transformed to the CS paradigm. Then, potential loss of interaction semantics can result in suboptimal or even problematic system integration. Moreover, when designing new applications integrating heterogeneous systems, the application developer has no access to their actual or fine-grained interaction semantics, since these are wrapped behind single-scheme interfaces.

To overcome the limitation of today's ESB-based connectors for cross-domain interoperability in the FI, we introduce a new connector type, called *GA connector*, which stands for "*Generic Application connector*". The proposed connector type is based on the service bus paradigm in that it achieves bridging across heterogeneous connector types. However, the behavior of the GA connector type differs from that of classical ESB connectors by bridging protocols across heterogeneous paradigms, which is further realized by paying special attention to the preservation of the semantics of the composed protocols. Indeed, the GA connector type is based on the abstraction and semantic-preserving merging of the common high-level semantics of base interaction paradigms. This solution shall serve rethinking the typical SOA- and ESB-based composition of heterogeneous distributed systems so as to meet the requirements of the Future Internet (FI).

This chapter details our systematic abstraction approach to interaction protocol interoperability across paradigms, which relies on the following definitions:

1) First, in Section 3.2, we introduce base CS, PS and TS connector types, which formally characterize today's core interaction paradigms. The proposed types comprehensively cover the essential semantics of the considered paradigms, based on a thorough survey of the related literature and representative middleware instances.

   Still, in a first step, we concentrate on abstracting discrete interactions supported by state of the art middleware protocols. However, continuous interactions are increasingly important in the Internet-connected world due to the exchange of content via richer media and further interaction with and within the Internet of Things. Continuous interactions will be studied in the next period and related evolution of the architectural style will be reported in Deliverable D1.4.

2) Second, in Section 3.3, we abstract further the three core connector types into a single Generic Application (GA) connector type, which provides an abstract union of the three types, thus preserving their interaction semantics. As a result, GA connectors support interactions among highly heterogeneous services of the FI, and especially across domains.

3) Third, in Section 3.4, we show how the GA connector implements cross-paradigm interoperability according to the semantics of the various paradigms that get composed.

   This paves the way toward automated reasoning about, and fostering of, interoperability between components/services at the middleware layer, with respect to their respective interaction paradigms. In a complementary way, the next chapter deals with interoperability at the choreography level (i.e., application layer) by accounting for the components' respective behavioral specification.

4) Finally, as discussed in Section 3.5, the CHOReOS connectors impact upon the definition of service-oriented components, as they leverage interaction paradigms that are not traditionally associated with SOA. We briefly analyze such impact at the end of this chapter, while related extension to the definition of CHOReOS component introduced in the previous chapter, will be detailed in the next Deliverable D1.4.

Prior to the specification of the CHOReOS connector types, the next section briefly recalls the main notion underlying the definition of a connector in an architectural style, in particular building upon previous publication by one of the deliverable authors as part of the CONNECT project[2] [37] and the text book by Taylor *et al.* [74].

3.1. Background on Connector Formalization

In the context of distributed systems, a connector abstracts a complex interaction behavior that is facilitated by middleware, which provides services to realize this interaction. In particular, middleware overcomes the heterogeneity of the distributed infrastructure by establishing a software layer that homogenizes the infrastructure's diversities using a well-defined and structured distributed programming model [38]. In particular, middleware induces an interaction paradigm for enabling distributed networked systems to coordinate [54].

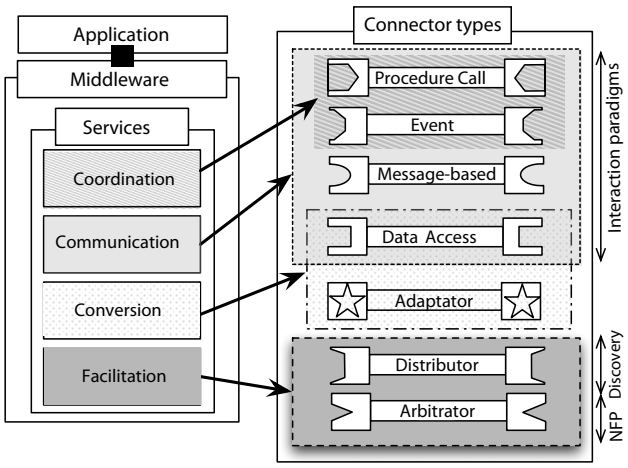


Figure 3.1: Middleware-based connector classification

Based upon the classification of connectors introduced in [48, 74], services provided by middleware are depicted in Figure 3.1. The *coordination* and *communication* services support the transfer of data and control among components and can be realized by different connector types, each of which defining an *interaction paradigm* such as *procedure call*, *event*, *message-based*, or *data access* connectors. The *adapter* connector type provides a *conversion* service to support interaction among heterogeneous components, while services that *facilitate* interaction among components are achieved using the *distributor* and *arbitrator* connector types. Distributor connectors perform discovery through the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths. Non-functional properties (NFP) are managed by *arbitrator* connectors that streamline system operations, resolve any conflict and redirect the flow of control.

In the context of the reported work, we more specifically concentrate on the definition of connector types with respect to the realization of interaction paradigms. We are further concerned with the definition of a new adapter connector type sustaining interoperability across heterogeneous interaction paradigms, i.e., the GA connector type. As for CHOReOS-specific distributor and arbitrator connector types, they will be examined in the next WP1 deliverable D1.4, based on WP3's 1st year results. Still, the interested reader may refer to D3.1 for a concrete instantiation of such connectors as part of the CHOReOS middleware.

Each connector type is associated with different dimensions (and subdimensions) representing its architectural details. For example, a procedure call connector defines the *Parameters* dimension that

[2] http://connect-forever.eu/.

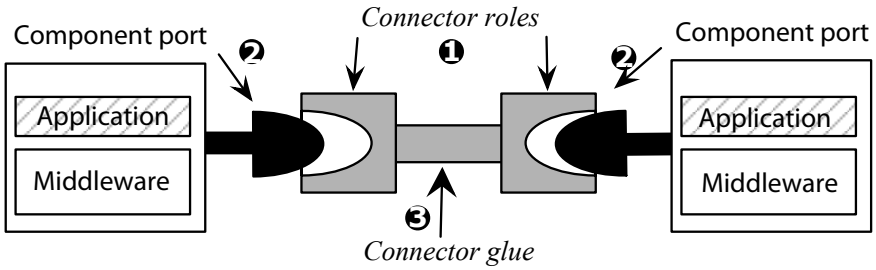


Figure 3.2: Components & Connector

is subdivided into *data transfer*, *semantics*, *return value*, and *invocation record* subdimensions. The procedure call connector type is also associated to other dimensions such as *Entry point* associated to two subdimensions, *single* or *multiple*, *Invocation* defining the *implicit* and *explicit* subdimensions, *Synchronicity*, *Cardinality*, and *Accessibility*. The values associated to the various dimensions and subdimensions define a connector implementation, that is, a specific middleware. For example, SOAP[3] (Simple Object Access Protocol), CORBA[4] (Common Object Request Broker Architecture), and RMI[5] (Remote Method Invocation) are specific middleware defining implementations of the procedure call connector type.

Formally and according to [2], the behavioral semantics of a connector is defined by a set of *role* processes and a *glue* process where:

- *role* processes (See Figure 3.2, ❶) specify the expected local behavior of each of the interacting parties.
- *glue* process (See Figure 3.2, ❸) specifies how the behaviors of these parties are coordinated.

In addition, the interaction protocols of components are specified by *port* processes (See Figure 3.2, ❷). Then, by specifying the behavior of roles, glues and ports using some process algebra, (e.g., the authors of [71] use FSP processes [45]), architectural matching and thus interoperability may be reasoned upon. Specifically, a component can be attached to a connector only if its port is *behaviorally compatible* with the connector role it is bound to. Allen and Garlan [2] define behavioral compatibility between a component port and a connector role based on the notion of refinement. Informally, a component port is behaviorally compatible with a connector role if the process specifying the behavior of the former refines the process characterizing the latter. In other words, it should be possible to substitute the role process by the port process.

Note that interaction protocols in general span multiple layers and in particular the application and middleware layers if we assume IP-based networking environments. Indeed, both layers are sources of heterogeneity. The behavior of a connector may then be defined as a hierarchical protocol that specifies the behavior of the application-layer interaction protocol in terms of middleware-specific protocols [37]. However, in this chapter, we concentrate on interoperability at the middleware layer, across heterogeneous middleware and further paradigms; interoperability at the application layer is addressed at the level of the overall choreography, which is the focus of the next chapter.

Figure 3.3 depicts our overall approach to interoperability across interaction paradigms. While networked services (*aka* networked applications) rely on legacy protocols and hence use their associated API to interact with their environment, these legacy protocols are mapped onto associated primitives of the GA connector toward sustaining interoperability in the FI without sacrificing the protocols' semantics. Then, internally to the GA connector, possible architectural mismatches are solved based on the pairwise matching of GA's input primitives with GA's output primitives, which account for the rich semantics

[3] http://www.w3.org/TR/soap/.
[4] http://www.omg.org/technology/documents/corba_spec_catalog.htm.
[5] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

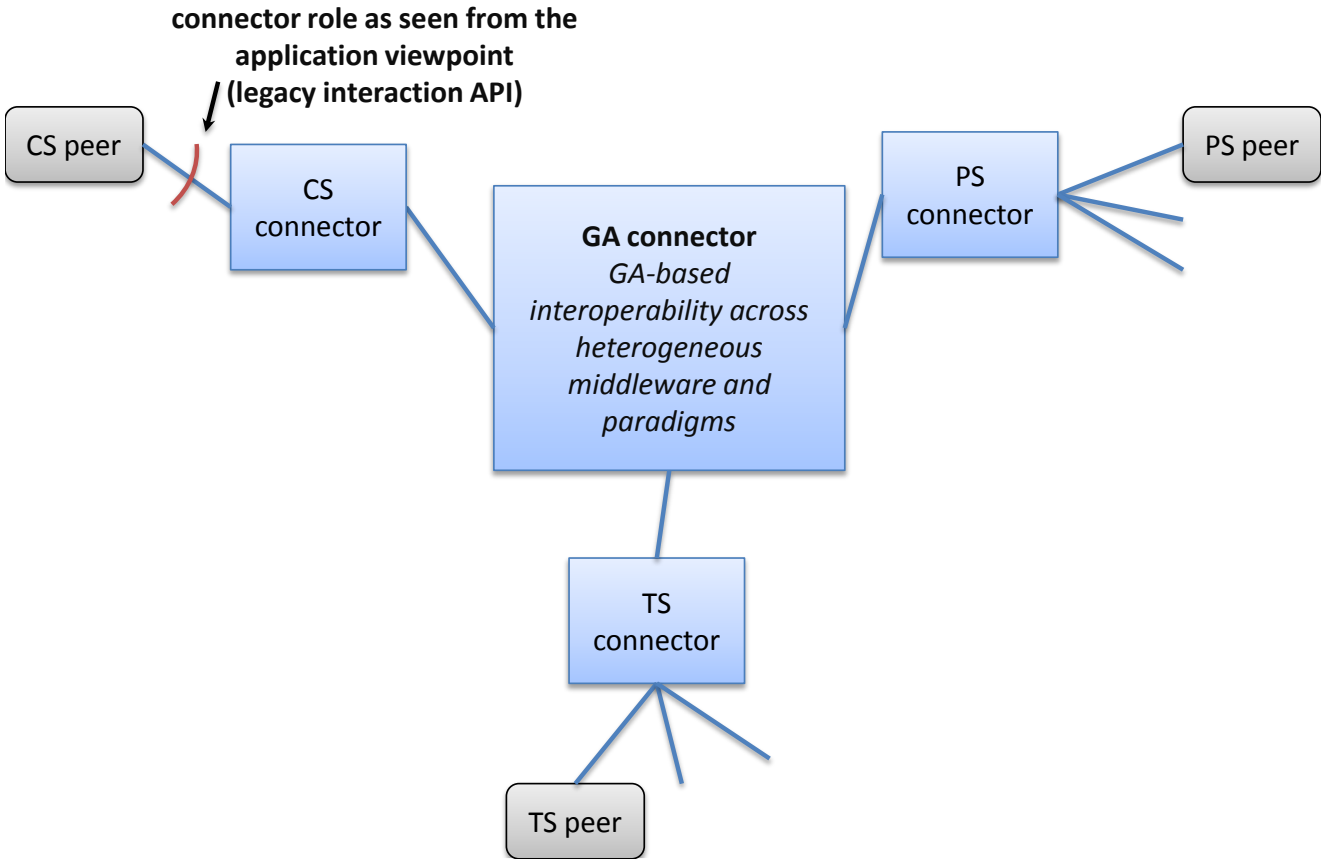


Figure 3.3: GA-based connector interoperability

of input and output actions that get mediated thereof. The next section introduces the base connector types associated with the definition of the core interaction paradigms encountered in today's distributed systems. Then, as already indicated, Section 3.3 introduces the CHOReOS-specific GA connector type, which allows for cross-paradigm interoperability based on the mapping of base connector types to/from GA as investigated in Section 3.4.

In the following sections, connector types are formally specified in terms of: (i) their API (Application Programming Interface) and (ii) their roles, i.e., the semantics of interaction of the connected component(s) with the environment via the connector. Regarding the latter, the behavioral specification of roles from a middleware perspective relates to specifying the production and consumption of information in the network, while the semantics of the information are abstracted and dealt with at the application layer. The behaviors of the connector roles are then specified using (the graphical representation of the corresponding) LTS (Labeled Transition Systems) that commonly serve defining the semantics of process algebra and further enable graphical representation of processes, where ! (resp. ?) denotes an output (resp. input action). The graphical representation of LTS being self-explanatory, we do not provide a detailed definition of LTS in this chapter; still, the interested reader may refer to Section 4.2.1 in the next chapter for detail. As for the connector glue, it defines how (in general a pair of) matching producer and consumer roles coordinate in the environment. Then, in the context of our work, we concentrate on the definition of the glue process implemented by the GA connector, which enables interoperability across highly heterogeneous services. On the other hand, the glue processes of base connector types are well known and rather direct given the specification of the role processes they coordinate. Thus, they are simply informally sketched in what follows.

3.2. Base Connector Types Abstracting Core Interaction Paradigms

This section identifies the three principal interaction paradigms used in distributed systems (i.e., CS, PS and TS), and defines the corresponding connector types. The proposed connector types are the outcome of an extensive survey of these paradigms as well as related middleware platforms in the literature. Our objective is to be able to abstract in each corresponding connector, a large number of interaction protocols, as implemented by today's middleware solutions, by a comprehensive set of semantics.

We further recall that the proposed connector types are to be extended in a second step to cope with continuous interactions, which are crucial in the context of the Internet of Things. This will be more precisely studied in the next period and reported upon in Deliverable D1.4.

3.2.1. Client-Server connector type

The *CS connector type* integrates a wide range of semantics, covering both the *direct (i.e., non queue-based) messaging* and *remote procedure call* (RPC) paradigms. It also enables all different kinds of reception semantics, such as blocking, blocking with timeout, or non-blocking.

The CS interaction paradigm is the most widely employed one, e.g., in middleware platforms like Web Services, Java RMI and CORBA. It imposes *space coupling* between the two interacting entities, i.e., at least the sending entity must know the receiving entity and hold a reference of it, as well as *time coupling*, i.e., both entities must be connected at the time of the interaction. A message carrying data is sent directly from the sending entity to the receiving entity. The receiving entity may choose to block its execution, synchronously waiting for the message (as long as it takes or with a timeout), or set up a callback function that will be triggered asynchronously by the middleware when the message arrives.

CS connector API: The above semantics are supported by the primitives and their arguments listed at the top of Figure 3.4, where the '&' symbol is used to indicate a return argument of a primitive. Precisely:

- *send* executes the synchronous emission of a message embedding *operation* name and related *input* parameter, to *destination*.
- *receive_sync* executes the synchronous reception of a single message, which may be waited for until the *timeout* expires.
- *receive_async* sets the asynchronous reception of multiple messages and specifies the associated *callback*.
- *notify* serves notifying the *callback* upon reception of relevant message by the middleware.
- *end_receive_async* closes the channel for asynchronous receipts.

Note that the traditional RPC semantics (aka *invoke* primitive) is realized from the viewpoint of the caller by combining the sending of a request message (via *send()*) and reception of a reply message (via *receive_sync()*) in a two-way synchronous interaction. The RPC interaction from the viewpoint of the callee can further be easily composed with a *receive()* (synchronous or asynchronous) and a subsequent *send()*.

Additionally, the proposed interface considers a single recipient for the message (i.e., *1-1* interaction), while it can be easily generalized to multiple recipients (i.e., *1-n* interaction). Although client-server systems traditionally implement 1-1 interaction schemes, *1-n* interaction schemes become relevant for group communication that is particularly suitable for enforcing non-functional dependability properties.

send(destination, operation, input)
receive_sync (& source, operation, & output, timeout)
receive_async (& source, operation, & output, callback)
notify(callback, source, output)
end_receive_async (callback)

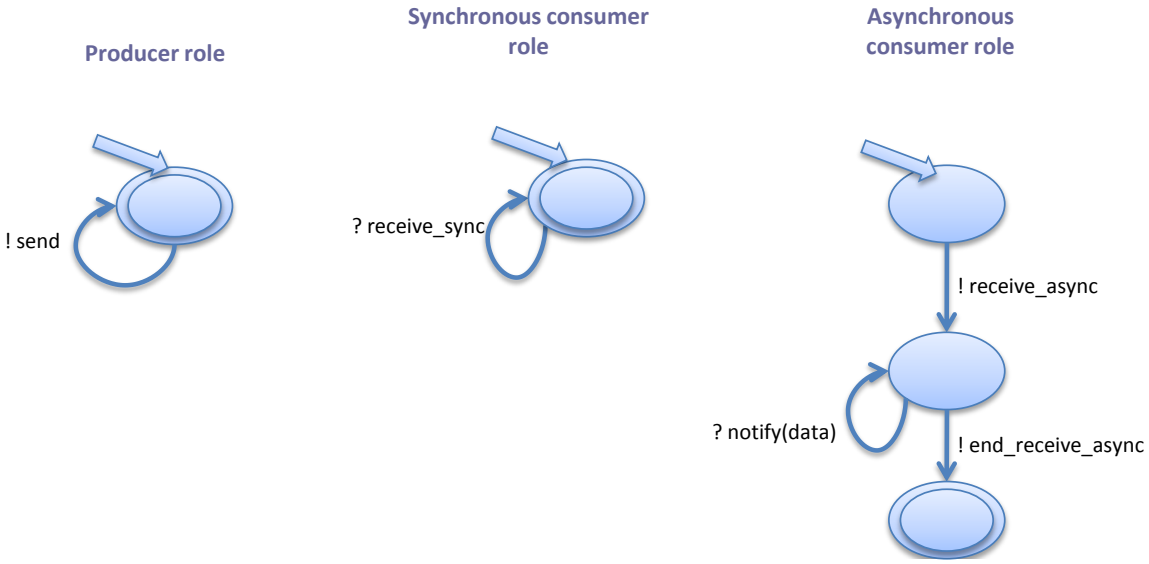


Figure 3.4: CS connector API (top) & Interaction semantics (bottom)

CS interaction semantics: The semantics of the CS connector regarding application (service) interaction with the environment is depicted at the bottom of Figure 3.4. Precisely, the figure specifies the behavior of the connector roles, which enable to produce and consume information in the networked environment using the connector API. In particular, note that the LTS actions are labeled according to the operations of the connector API that they abstract. While production is synchronous, consumption may be either synchronous and asynchronous under the CS paradigm. Regarding the behavior of the connector glue that is omitted here, it consists in synchronizing the send (*send*) of the producer role with the synchronous (*receive_sync*) or asynchronous receipt (*notify* given *receive_async* has been executed before) of the matching consumer role. In the case of asynchronous consumption, the synchronization is mediated *via* the middleware, which triggers the appropriate callback.

3.2.2. Publish-subscribe connector type

The *PS connector type* abstracts in a comprehensive way the different types of publish/subscribe systems, such as *queue-*, *topic-* and *content-based systems* [29]. It further enables rich reception semantics: synchronous pull by the subscriber, which may be blocking, blocking with timeout, non-blocking; or asynchronous push by the broker.

In the PS interaction paradigm, multiple peer entities interact *via* an intermediate *broker* entity. Publishers produce events, which are received by peers that have previously subscribed for receiving the specific events. In *topic-based PS* [51], events are characterized with a topic, and subscribers subscribe to specific topics. In *content-based PS* [21], subscribers provide content filters (conditions on specific attributes of events), and receive only the events that satisfy these conditions. Following the common practice for related middleware platforms [51], we also integrate in our PS model *queue-based messaging*. In this case, an event is sent from a publisher to the queue of a specific subscriber, which may be considered as a constrained case of topic-based PS.

The PS paradigm enables *space decoupling* between interacting peers: peers do not know each other or how many they are; with the exception of queue-based PS, where at least the publisher holds a reference of the subscriber. Moreover, the PS paradigm enables *time decoupling*: peers do not need to be present at the same time, subscribers maybe disconnected at the time of the interaction; they can receive the pending events when reconnected. The broker maintains an event until all related subscribers have received it or until the event expires. Subscribers may choose to check for pending events synchronously themselves (just check instantly or wait as long as it takes or with a timeout) or set up a callback function that will be triggered asynchronously by the broker when an event arrives.

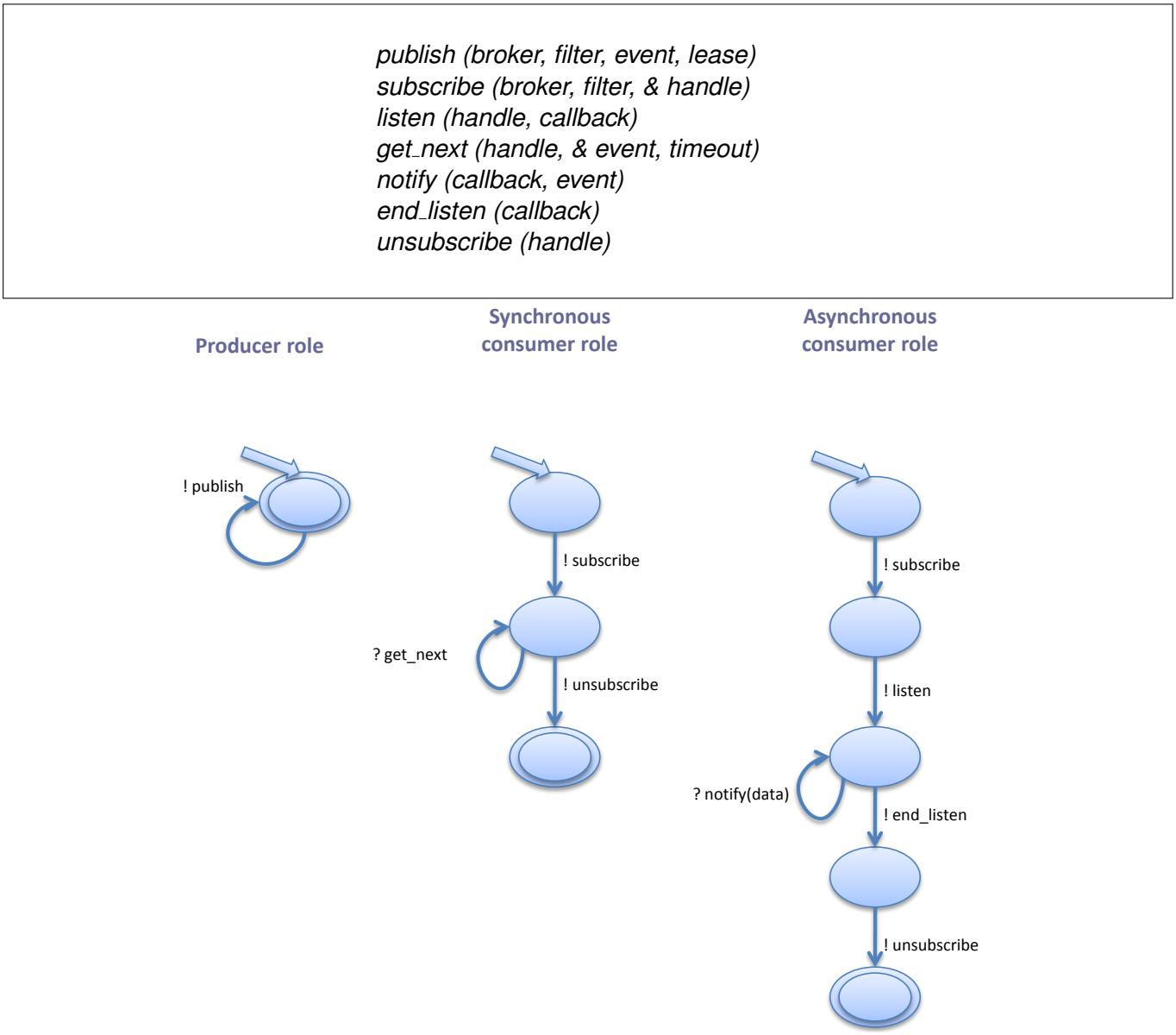


Figure 3.5: PS connector API (top) & Interaction semantics (bottom)

PS connector API: The primitives associated with the above interaction semantics are listed at the top of Figure 3.5, where we represent the notions of queue, topic and content with the generic *filter* parameter. In addition, the *lease* parameter stands for the lifetime of the event. More precisely, the PS connector type implements the following primitives:

- *publish* publishes an event.

- *subscribe* subscribes for receiving events.
- *listen* enables asynchronous reception of multiple events.
- *get_next* executes synchronous reception of a single event.
- *notify* is called by the middleware *broker* to notify an event to its subscribers.
- *end_listen* closes a channel for event notification.
- *unsubscribe* unsubscribes from receiving events.

PS interaction semantics: The specification of the consumer and producer roles associated with the PS connector is depicted at the bottom of Figure 3.5. Same as for the CS paradigm, the production of information is synchronous although decoupled from the consumer in time and space through the middleware broker. The consumption may be either synchronous or asynchronous. Finally, the coordination between matching producers and consumers achieved by the connector glue is realized via the middleware broker.

3.2.3. Tuple Space connector type

Regarding the base connector type abstraction for data-oriented interactions, we build on the tuple space paradigm although more specialized than basic shared memory. This is to model rich interaction semantics that are now associated with shared data space in distributed systems.

The definition of the *TS connector type* is based on the classic tuple space semantics as introduced by the Linda coordination language [32], while it further incorporates a number of advanced features that have been proposed in the literature, such as asynchronous notifications, explicit scoping, and bulk data retrieval primitives.

In the TS interaction paradigm, multiple peer entities interact via an intermediate *shared data space*. Peers can post data into the space and can also synchronously retrieve data from it, either by taking a copy or removing the data. Data take the form of tuples; a tuple is an ordered list of typed elements. Data are retrieved by matching based on a tuple template, which may define values or expressions for some of the elements. Same as the PS paradigm, the TS paradigm enables both *space and time decoupling* between interacting peers. Nevertheless, TS has a number of specifics. In particular, peers have access to a single, commonly shared copy of the data. Also, peers do not need to subscribe for data, they can retrieve data spontaneously and at any time.

The data space maintains data until they are removed by some peer or until the data expire. Additionally, concurrency semantics of the data space are non-deterministic: among a number of peers trying to access the data concurrently, the order is determined arbitrarily. In addition to the above semantics, our TS model integrates the following extensions. First, besides synchronous retrieval of tuples, a number of approaches have enabled asynchronous notifications in tuple spaces [17, 31]. More specifically, peers may choose to check for matching tuples synchronously themselves (just check instantly or wait as long as it takes or with a timeout) or set up a callback function that will be triggered asynchronously by the data space when matching data appear. This call does not carry the data, possible action of taking or reading the data should be executed by the peer. Second, in distributed realizations of tuple spaces and especially those enabled in mobile environments [52] and/or location-aware environments (e.g., wireless sensor networks) [9], it is important to be able to identify and access only a part of the shared space; this is commonly denoted by the concept of scoping in tuple spaces. Third, several authors have pointed out the *multiple read* problem [69]: if there are multiple tuples matching a read request issued to the tuple space, the tuple retrieved is selected arbitrarily; thus, a sequence of read requests does not guarantee that all the matching tuples will be retrieved. Some approaches propose *bulk read* primitives [52, 68] that retrieve all the matching tuples.

TS connector API: Primitives of the TS connector are listed at the top of Figure 3.6 and relate to:

- *out* inserts the given tuple for a max *lease* period.
- *take* executes asynchronous reception and removal of a single or all matching tuples, depending on the *policy* specification (which may be *one* or *all*), until *timeout* expires.
- *read* has similar semantics as *take* but does not remove the tuple(s).
- *register* sets up asynchronous notification for new tuples.
- *notify* is called by the tuple space system/middleware when a new tuple appears.
- *unregister* closes a notification channel.

In the above primitives, the parameter *scope* may be a value or an expression that resolves to a specific part of the shared space.

out (tuple_space, scope, template, tuple, lease)
take (tuple_space, scope, template, policy, & tuple, timeout)
read (tuple_space, scope, template, policy, & tuple, timeout)
register (tuple_space, scope, template, callback)
notify (callback)
unregister (callback)

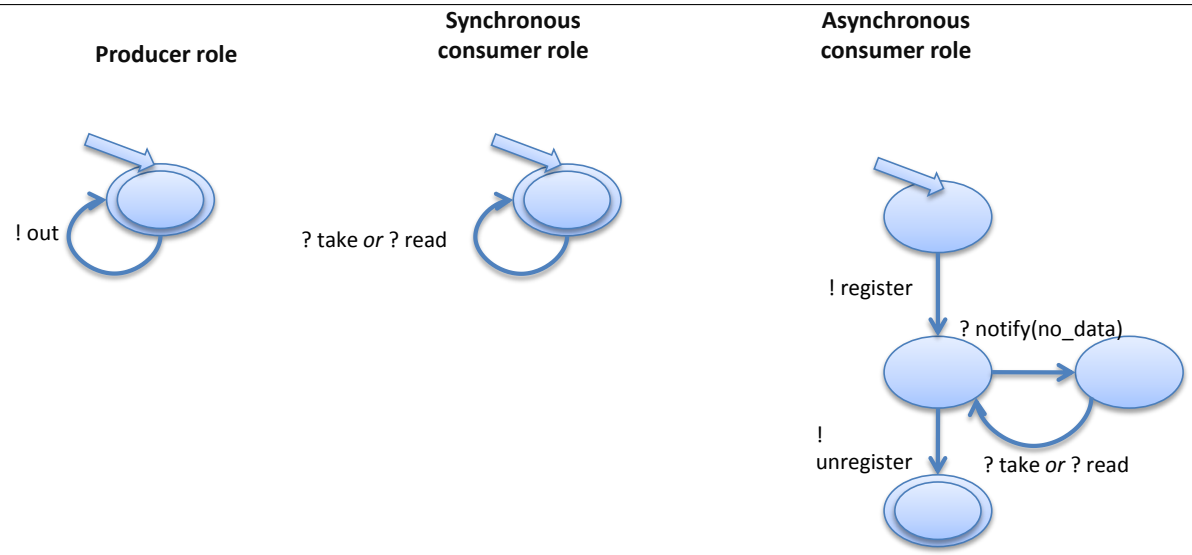


Figure 3.6: TS connector API (top) & Interaction semantics (bottom)

TS interaction semantics: The specification of the consumer and producer roles associated with the TS connector is depicted at the bottom of Figure 3.6. Again, the production of information is synchronous (with the shared space) while its consumption may be either synchronous or asynchronous. The coordination implemented by the connector glue relies on the tuple space maintained by the middleware.

3.3. Generic Application Connector Type

Given the three base middleware-layer connector types defined in the previous section, we now introduce the CHOReOS Generic Application (GA) connector type. Our objective is to devise a single generic connector that comprehensively incorporates the end-to-end interaction semantics of application entities that employ any of the three former base (middleware) connectors.

With respect to the service bus paradigm that has already proved successful toward sustaining interoperability in SOA, our goal is to introduce an intermediary reference protocol (e.g., the ESB pivot protocol such as JBI used in the PetalS ESB) that leverages the richness of today's interaction paradigms, as opposed to constraining interaction types to a single (principally CS) paradigm. Further, as already stressed, it is central that the proposed pivot protocol allows for cross-paradigm interoperability.

We identify two principal high-level primitives for service interaction with the environment:

1) A *post()* primitive employed by a peer for sending data to one or more other peers (i.e., production of information), and

2) A *get()* primitive employed by a peer for receiving data (i.e., consumption of information).

For example, a PS *publish()* primitive can be abstracted by a *post()*.

Then, following the definition of the base connector types in the previous section, a producer/post - consumer/get end-to-end interaction may be characterized by one of the following four types of coupling between the sender (producer) and the receiver(s) (consumer):

- **Strong coupling**: Implies *space and time coupling* between peers. Receiving peer(s) receive each a dedicated copy of the sent data. This coupling corresponds to the CS model.
- **Weak coupling**: Implies *space coupling* (for queue-based PS) or *decoupling* (for topic-/content-based PS)) and *time decoupling* between peers. Receiving peer(s) receive each a dedicated copy of the sent data. This coupling corresponds to the PS model.
- **Very weak coupling**: Implies *space and time decoupling* between peers. Receiving peer(s) have access to a commonly shared copy of the data. They may manifest their intention to receive the data also after the data have been sent. A receiving peer may receive the sent data, depending on: (i) individual/common policies of peers for accessing the data; and (ii) non-deterministic concurrency when accessing the data. This coupling corresponds to the TS model.
- **Any coupling**: Implies that any of the above three types of coupling may apply. This enables developing applications that are tolerant to any underlying middleware interaction protocol, although practical implication with respect to service development is yet to be investigated.

GA connector API: The complete set of GA primitives is given at the top of Figure 3.7. Two parameters need to be explained in the introduced primitives:

1) First, we have introduced the explicit scoping parameter *scope* to generalize addressing for the different types of coupling. We analyze *scope* as {*mainscope, subscope*}. Then, for the CS model, this maps to {*destination/source, operation*}, for PS, it maps to {*broker, filter*}, and for TS, to {*tuplespace,* {*scope, template*}}. In practice, *scope* enables restricting the entities that have access to the data conveyed in a sender/post - receiver/get interaction (it also specifies partial semantics for the data). For example, we map the *destination* parameter of CS *send()* to *mainscope*; thus, the *input* data carried by *send()* will be accessed only by the single CS destination peer (in another example, the *filter* parameter of PS mapped to *subscope* specifies both addressing and partial semantics of an event).

post (coupling, scope, data, lease)
set_get (coupling, [&] scope, & handle)
get_async (handle, & data, callback)
get_sync (handle, policy, & data, timeout)
notify (callback, scope, data)
end_get_async (callback)
end_set_get (handle)

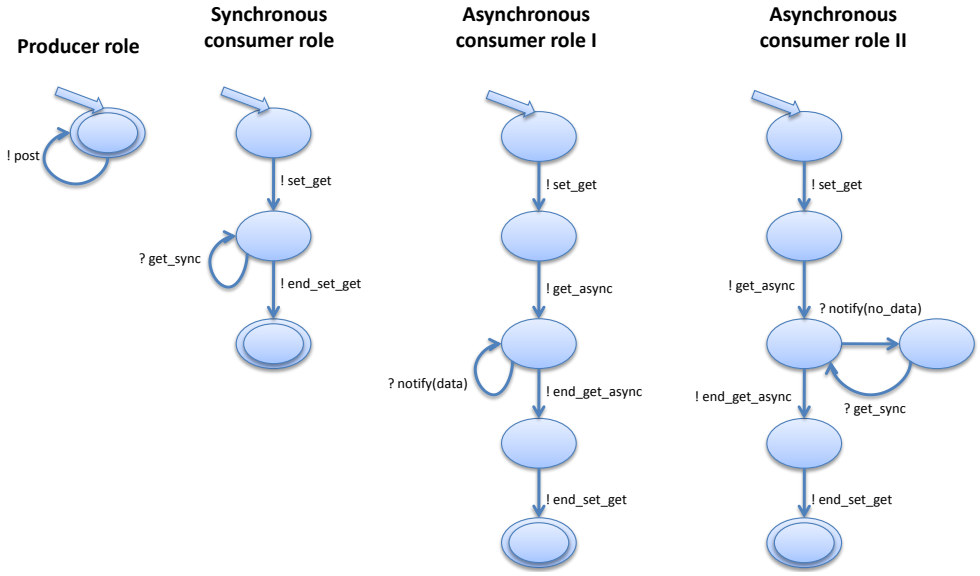


Figure 3.7: GA connector API (top) & Interaction semantics (bottom)

2) Secondly, we have introduced *policy* to further specialize the *get_sync* primitive in the case of very weak coupling (TS model). Thus, *policy* may be *remove*, *copy*, *remove_all*, *copy_all*, which correspond to the various TS data retrieval primitives. For strong or weak coupling, *policy* is *remove* by default.

Precisely, the GA primitives are as follows:

- *post* produces a *data* in the networking environment according to the semantics set by the parameters *coupling* (i.e., *strong*, *weak*, *very_weak* and *any* as defined above), *scope* (i.e., defining *mainscope* and *subscope*), and *lease* (i.e., lifetime of the *data*).
- *set_get* sets up reception resources at the connector (middleware), where the '[&]' symbol is used to indicate that an argument of a primitive may be an input or a return argument, depending on the context of use.
- *get_async* enables asynchronous reception of multiple pieces of data.
- *get_sync* executes synchronous reception of a single piece of data.
- *notify* enables asynchronous reception, and is called by the middleware.
- *end_get_async* closes a channel for asynchronous reception.
- *end_set_get* closes a reception channel.

GA interaction semantics: The behaviors of the GA connector roles regarding consumption and production of information, are specified at the bottom of Figure 3.7. The production of information is synchronous, although space and time decoupling with the consumer is possible, depending on the actual parameters of *post*. And, as for base connectors, we distinguish between synchronous and asynchronous consumption of information. Additionally, we distinguish two forms of asynchronous consumption, depending on whether the consumption is in the *push* (role I) or *pull* (role II) mode.

The above presented GA connector model provides an abstract union of the base CS, PS and TS connector types, thus preserving by construction their interaction semantics. Furthermore, our complete two-step abstraction approach, i.e., from middleware platforms to the CS/PS/TS connector types, and from the latter to the GA connector type, provides a concise high-level API that is agnostic to underlying middleware platforms. We employ these desirable features for integrating heterogeneous systems in the next section.

3.4. Interoperability Across Heterogeneous Interaction Paradigms

According to the service bus paradigm that is adopted within CHOReOS, interoperability relies on the mapping of legacy middleware protocols to/from the bus' pivot protocol. More precisely, legacy middleware protocols are first abstracted through the base connector types introduced in Section 3.2, where it is rather straightforward to map legacy middleware protocols to the corresponding base connector types. In particular, state of the art ESBs like PetalS implement the mapping of legacy (mostly CS-based) middleware protocols to the bus' pivot protocol that is an instance of the CS connector type. Then, as already stated, the GA connector type defines a pivot protocol for the CHOReOS service bus that makes interoperable (at the middleware layer) the highly heterogeneous services of the FI.

The following section defines the mapping of the roles implemented by the base connector types to/from the corresponding roles of the GA connector type. In addition, fostering interoperability across heterogeneous connector types requires defining appropriate mapping between consumer and producer roles that may differ with respect to time and space coupling, and scoping. This relates to the definition of the glue process implemented by the GA connector, which is the focus of Section 3.4.2.

3.4.1. Mapping to and from GA

Based on the definition of the GA primitives in Section 3.3, the mapping between the GA API and roles, and the corresponding APIs and roles of CS, PS and TS is quite direct:

- Figure 3.8 gives the mapping of GA parameters to/from corresponding base protocol parameters, with the coupling parameter of the base protocols being implicitly defined by the protocols' respective semantics (i.e., CS=strong, PS=weak and TS=very_weak coupling).
- Figure 3.9 depicts the mapping of GA roles (leftmost side of the figure, where the two asynchronous consumer roles of GA are merged in a single LTS) to/from base protocol roles. In the mapping, ϵ actions are introduced in role processes that do not support the corresponding GA actions; in practice, this means that the specific GA actions are produced or consumed internally by the GA connector (i.e., middleware).

3.4.2. Cross-paradigm interoperability: GA glue

The GA glue must ensure proper coordination of semantically matching producer and consumer despite heterogeneity of the actual protocols executed by the services. As already mentioned, this is directly supported by state of the art ESB solutions when the producer and consumer both instantiate CS connector roles. However, we want to get a step further by enabling heterogeneous producer and

GA	CS	PS	TS
Coupling (= strong/weak/very_weak)	-	-	-
scope.mainscope	destination/source	broker	tuple space
scope. subscope	operation	filter	{scope, template}
handle	-	handle	-
callback	callback	callback	callback
policy	-	-	policy
data	input/output	event	tuple
lease	-	lease	lease
timeout	timeout	timeout	timeout

Figure 3.8: Mapping GA parameters to/from base (CS, PS, TS) parameters

consumer to interoperate as long as they exchange semantically matching information/data although possibly using distinct interaction paradigms. In other words, we want to enable a *get* and a *post* actions of GA to coordinate if they *semantically match* in terms of the application semantics they carry although mismatching from the standpoint of the interaction paradigms they abstract. Note that the semantic matching of application data (messages) is dependent upon the application layer and is thus assumed to be reasoned upon and possibly mediated at that layer (i.e., at the level of choreography and related coordination delegates discussed in the next chapter). Further, we do not study the syntactic translation of middleware messages from one middleware technology to another, as this is abstracted at the software architecture level. Such an issue is more specifically addressed in WP3 that develops the CHOReOS middleware, which shall enable the CHOReOS architectural style and in particular associated interoperability across interaction paradigms in the FI.

Figure 3.10.*a)* (resp. *b)*) analyzes the semantics of the GA *post* (resp. *get*) action under the various couplings that are supported. However, we ignore the *any coupling* semantics since it semantically matches with any of the other couplings and thus trivially supports interoperability. Then, the GA *glue* shall realize the necessary mediation between heterogeneous coupling semantics for a data message to be properly exchanged between the sender (*post-er*) and its semantically matching receiver(s) (*get-er(s)*). Figures 3.11 to 3.13 informally depict the coordination semantics implemented by the GA glue according to the respective semantics of the get-er and post-er, and hence across heterogeneous interaction paradigms. For all cases, coordination may occur only if the coordination actions of the get-er and post-er intersect in *space* and *time*. Overlap in space relates to the definition of the respective scopes by the get-er and post-er, which set the peer(s) that may get the data message via the GA connector. As for overlap in time, it depends on the time at which the coordination actions are executed by the get-er and post-er, and associated timeout and lease, as detailed in Figures 3.11 to 3.13. For instance in Figure 3.11, a get-er under strong coupling synchronizes successfully in time with a post-er under strong or weak coupling, if the post action is executed inside the time interval that the get-er is active (e.g., after get_sync has started and before it times out). On the other hand, if the post-er is under very weak coupling, the post action may be executed also in advance, but in any case the posted data should not expire before the get-er gets activated.

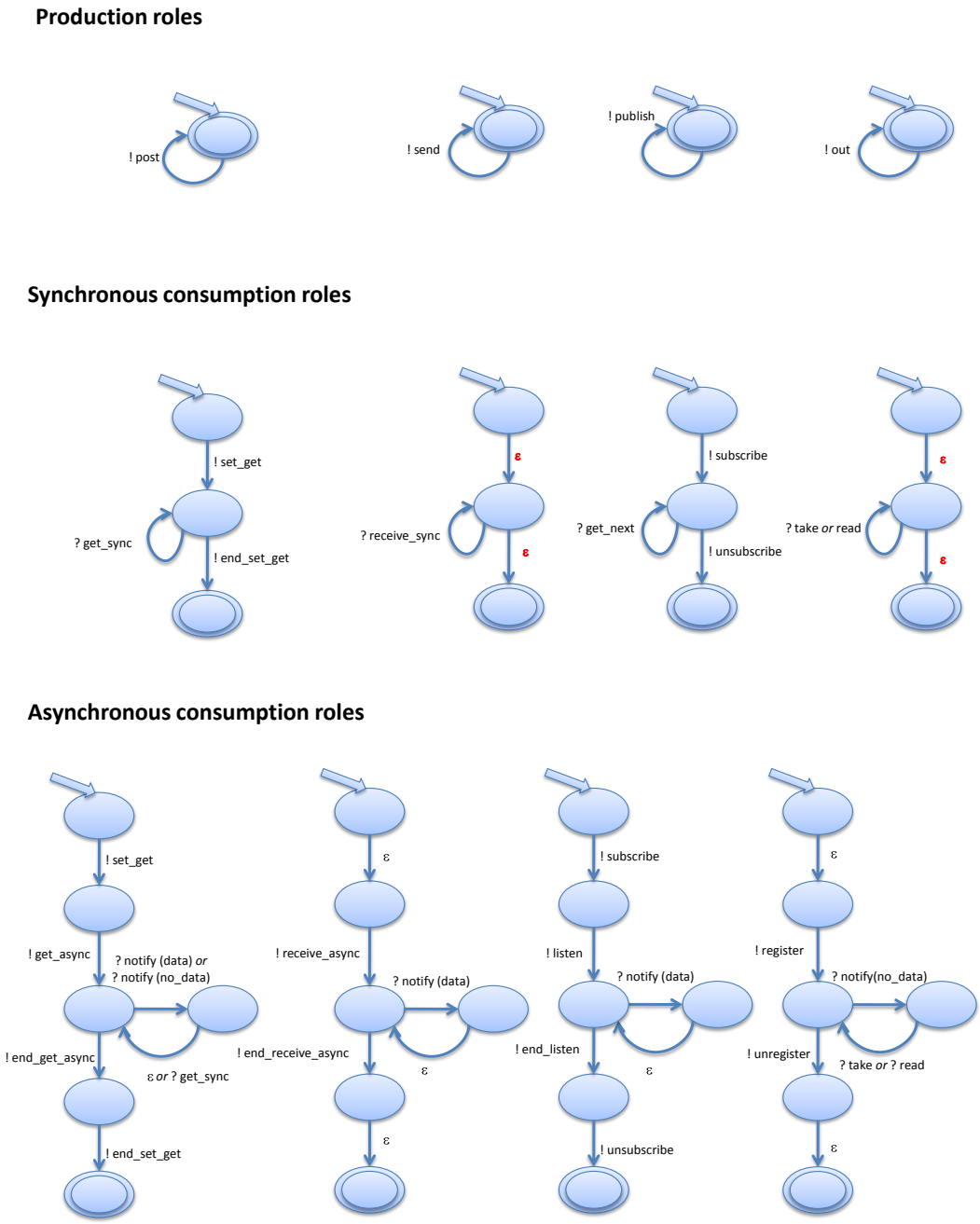


Figure 3.9: Mapping GA roles to/from base (CS, PS, TS) connector roles

The proposed coordination semantics of GA has still to be formally specified in order to thoroughly analyze the impact upon the coordinated systems. This will be undertaken during the next period and will be reported upon in Deliverable D1.4. In a complementary way, while this chapter has focused on the abstract specification of the CHOReOS connectors and especially the heterogeneous interaction paradigms that need to co-exist and further interoperate in the Future Internet, the CHOReOS middleware shall support those paradigms. This is specifically the focus of WP3 RTD work, which leverages and further enriches existing solutions to instantiate base connector types. Moreover, WP3 revisits the ESB paradigm upon developing the GA connector type to allow for interoperability across interaction paradigms. The interested reader is referred to Deliverable D3.1 for further detail about the CHOReOS service-oriented middleware for the FI.

	post
strong coupling	Post a *data* toward peer(s) of the given *scope*. Peer(s) should be online and have timely manifested its(their) interest in order to eventually get the *data*.
weak coupling	Post a *data* toward peers of the given *scope*. Peers can be offline (max *lease* time); however, they should have timely manifested their interest in order to eventually get the *data*.
very_weak coupling	Post a *data* toward peers of the given *scope*. Peers may be offline and have even late manifested their interest (max *lease* time) in order to may be get the *data*.

a) Post action

	get
strong coupling	The peer eventually gets the *data* if online and has timely manifested its interest.
weak coupling	The peer eventually gets the *data*, even if currently offline (max *lease* time), however having timely manifested its interest.
very_weak coupling	The peer may get the *data*, even if currently offline and/or having manifested a late interest (max *lease* time).

b) Get action

Figure 3.10: Coupling semantics of the *post* and *get* actions

3.5. Discussion: Impact of CHOReOS Connectors on the Overall Architectural Style

This chapter has introduced the connector types associated with the rich interaction paradigms that are encountered in today's distributed systems and thus need to integrate in the FI. However, this definition is initial as it is focused on discrete interactions, while continuous interactions also play a major role in the FI, especially in the Internet of Things sub-domain. In summary, although the definition of the CHOReOS connector types is based on a thorough survey of the state of art, it may still evolve according to further analysis of the IoT domain that is evolving at a fast pace. The proposed definitions are also subject to evolution based on the formal specification of the connector types and especially the connector glues that is ongoing work.

Another consideration related to the definition of the CHOReOS connectors is the impact on the overall CHOReOS architectural style, and hence on the definition of CHOReOS components and choreography-based coordination protocols. Impact on the latter is already accounted for in the next chapter, which assumes the GA connector type for the interaction of coordination delegates with services of the FI. Still, the diversity of connector types needs to be taken into account in the modeling of choreographies, which is examined in WP2 and builds on the BPMN standard notation. Indeed, access to services specified within CHOReOS choreographies may be realized via any of the CHOReOS connectors, including GA for greater flexibility in the FI. This further impacts on the possible interfaces for services, while Chapter 2 primarily concentrates on leveraging traditional CS services in a first step. Extension of the definition of the CHOReOS component types to account for the diversity of services *connect-able* via the CHOReOS connectors will be investigated in the next period and reported in Deliverable D1.4. Still, as a first step, Figures 3.14 to 3.16 give the key elements associated with the interfaces of PS-, TS- and GA-based components, where the meaning of given elements is direct from the semantics of CHOReOS connectors introduced in this chapter. Overall, such definition complements and hints on the evolution of the definition of service interface given in Section 2.1.3, which is under study.

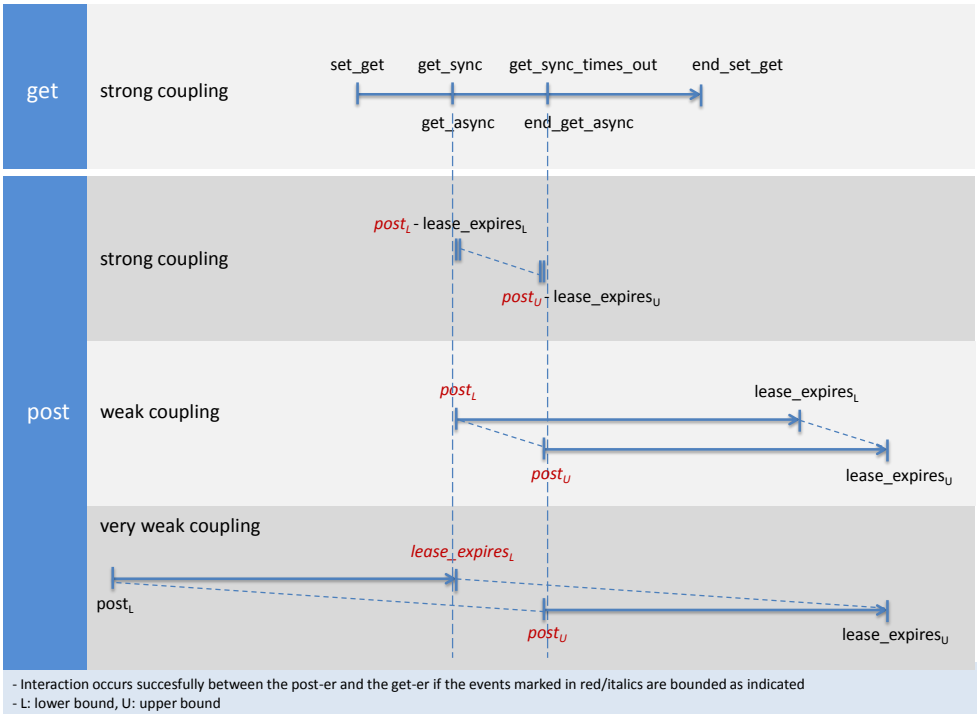


Figure 3.11: GA Coordination semantics (glue with get under strong coupling)

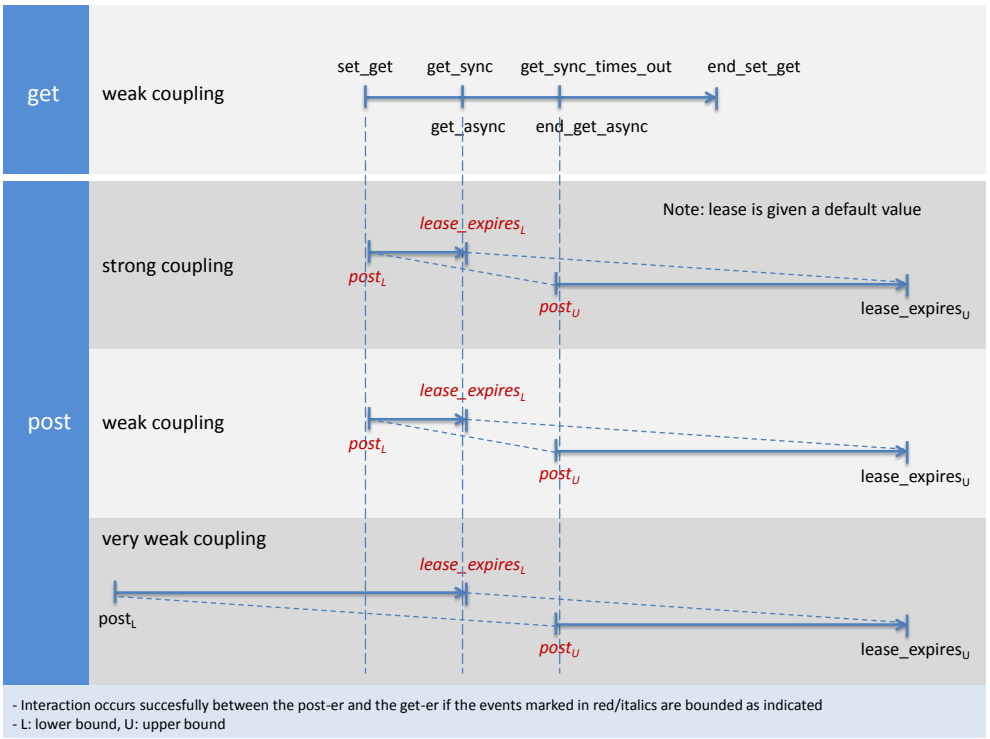


Figure 3.12: GA Coordination semantics (glue with get under weak coupling)

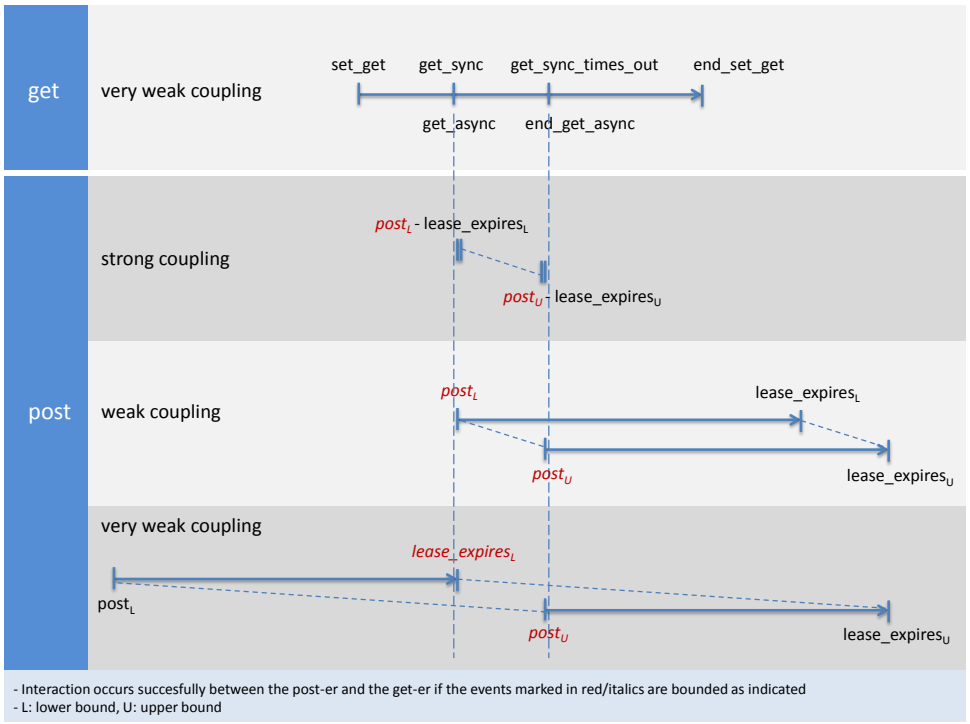


Figure 3.13: GA Coordination semantics (glue with get under very weak coupling)

PS-based service interface		
element	sub-element	attribute
event		semantics
		name
		type
		value
mainscope (of event)	publish-subscribe system identity	name
		address type
		address value
subscope (of event)	filter	semantics; part of semantics is {queue, topic, content}
		name
		type
		value
coordination semantics (of event)	produce/consume	{publish, subscribe}
	synchrony	{sync, async}
	lease	type
		value

Figure 3.14: Service interfaces related to PS interactions

TS-based service interface		
element	sub-element	attribute
tuple		semantics
		name
		type
		value
mainscope (of tuple)	tuple space system identity	name
		address type
		address value
subscope (of tuple)	scope	semantics
		name
		type
		value
	template	semantics
		name
		type
		value
coordination semantics (of tuple)	produce/consume	{out, take, read}
	consume policy	{one, all}
	synchrony	{sync, async}
	lease	type
		value

Figure 3.15: Service interfaces related to TS interactions

GA-based service interface		
element	sub-element	attribute
coupling		{strong, weak, very_weak}
data		semantics
		name
		type
		value
mainscope (of data)		semantics
		name
		type
		value
subscope (of data)		semantics
		name
		type
		value
coordination semantics (of data)	produce/consume	{post, get}
	consume policy	{remove, copy, remove_all, copy_all}
	synchrony	{sync, async_data, async_nodata}
	lease	type
		value

Figure 3.16: Service interfaces related to GA-based interactions

4 CHOReOS Coordination Protocols: Abstracting Choreography Behavior

As pointed out in the previous chapters, CHOReOS envisions a ubiquitous world of available services that can be discovered within the FI and *choreographed* to fit users' needs. Services play a central role in this vision as effective means to achieve interoperability between heterogeneous parties of a business process and independence from the underlying infrastructure. At the same time they offer an open platform to build new value added service-based systems as a collaboration of available services.

In this setting, different scenarios can be distinguished and different problems must be solved when actually realizing choreography-based systems [16, 18, 46, 49, 12, 60, 70, 72, 35, 10]. Some of these scenarios are described below:

- *Ad hoc*: Participant services are specifically developed to "fulfil" the specified choreography.

  Problem to be solved: implement the services correctly.

- *Role-based*: Services are proposed as participants "fitting" the role(s) to be played for the specified choreography.

  Problem to be solved: for each service, perform a conformance check between the service implementation and the model of the expected service role as extracted/projected from the choreography specification.

- *Goal-oriented and Requirements-based*: Existing services are discovered out of a goal-oriented and requirements-based choreography specification as "potentially suitable" participants. When saying potentially suitable we mean that the specified choreography can be realized, among all the possible collaborations, as a specific collaboration of the discovered services.

  Problem to be solved: coordinate the global interaction behavior of the participant services in order to guide their collaboration so as to fulfill the specified choreography.

Clearly, in general, the above scenarios are not disjoint in the sense that they jointly apply in large scale and heterogeneous environments as in CHOReOS. To some extent, the third scenario can be considered as the most general one yet possibly including also the others.

In this chapter, accounting for the definitions of CHOReOS *component* and *connector* of the previous chapters, we conclude the definition of the (preliminary) CHOReOS architectural style by adding the notion of CHOReOS *coordination protocol* that abstracts choreography behavior. Specifically, the CHOReOS *coordination protocol* introduces a higher, application-layer connector that defines system-wide behavior, based on the connection of CHOReOS components through middleware-layer connectors introduced in the previous chapter. We define specific architectural constraints to be imposed on the choreography-based system to suitably coordinate the discovered services, hence realizing the specified choreography. In particular, these constraints enable automated synthesis (to be worked out in WP2 during the second year of the project) of the coordination protocol "implied by" the specified choreography.

This chapter is structured as follow. Section 4.1 defines the problem of choreography-based coordination in the FI as a synthesis problem. Section 4.2 provides both an overall picture of the CHOReOS architectural style and suitable formal abstractions for choreography-based coordination.

4.1. Choreography-based Coordination in the FI

Before describing the problem of choreography-based coordination in the FI, we briefly introduce background concepts needed for a full understanding of our work on choreography-based coordination. The discussion has been kept as light as possible in order to give a good intuition to the reader without loosing his/her attention. Detailed formalisms and definitions are then referred to Section 4.2.

A choreography can be seen as a network of collaborating services, i.e., CHOReOS components, $S = \{S_1, \ldots, S_n\}$, that can simultaneously run. Hereafter, we will simply use the terms "component" or "service" to mean "CHOReOS component". It is worth to mention that a CHOReOS component can represent either a service that is developed to specifically exploit the facilities offered by the CHOReOS infrastructure or an existing (third-party) service. We recall that CHOReOS components can externally expose *provided* and/or *required* interfaces hence acting as *providers* and/or *consumers*, respectively. A CHOReOS component that acts as both a provider and a consumer is also referred to as a *prosumer*. As described in Chapter 3, components communicate/interact by means of the CHOReOS connectors, which implement various communication/interaction paradigms that are abstracted by the GA connector type for the sake of interoperability in the FI. By referring to Chapter 2, we recall that a component behavioral specification describes the set of all possible sequences of operations performed while interacting with the other components.

In this setting, the notion of coordination protocol is crucial since it might be the case that the collaborating services, although potentially suitable in isolation, when interacting together can lead to undesired interaction. That is, although a given set of services, if coordinated in the right way, can be used to achieve the specified choreography's goal and requirements, they can completely miss the goal or the requirements if coordinated in a different way. In order to deal with this problem, we use additional software entities and, at architectural level, interpose them among the services participating to the specified choreography. The intent of these additional entities is to coordinate the interaction of the participant services in a way that the resulting collaboration complies with the desired choreography.

In the CHOReOS architectural style, these coordinating entities are called *coordination delegates*. According to the specified choreography, they supervise and coordinate the collaboration of the participant services that are functionally and non-functionally abstracted as CHOReOS components. As detailed later, this is done by relying on the GA connector type, which connect the CHOReOS components and enable the communication among them. Thus, the coordination delegates perform "pure coordination", at application layer, by assuming an abstract description of the externally observable behavior of the participant services and considering heterogeneous communication already mediated by means of the GA connector (middleware layer). Moreover, by relying on the service's functional and non-functional abstractions described in Chapter 2, possible adaptation at the service interface level is already solved.

Concerning the CHOReOS purposes, we consider large scale and distributed computing/networking environments, in which it is not always possible or convenient to introduce centralized coordination logic. For example, it is not convenient to introduce an additional component which coordinates the interaction flow in a centralized way. Moreover, the coordination of several services might cause bottlenecks, hence slowing down the response time of the overall collaboration of the participant services. Conversely, distributing the coordination logic extends the possibility of actually realizing choreography-based systems to the FI large-scale contexts. This is why in our architectural style the coordination logic is *distributed* into a set of coordination delegates (that cooperate to support the realization of the specified choreography).

In CHOReOS, our idea is to consider the *choreography-based coordination problem* as a choreog-

raphy synthesis problem where service-oriented systems are assembled as service choreographies by following a well defined architectural style. The latter imposes constraints on the architecture of the service-oriented and choreography-based system so as to support, from a set of service behavior specifications, the automated synthesis of a set of coordination delegates. These delegates distributively cooperate to support the realization of the specified choreography.

The problem we want to treat can be phrased as follows: *Given a choreography specification C and a set of CHOReOS components $S = \{S_1, \ldots, S_n\}$, derive (when possible) a set of distributed coordination delegates $CD = \{CD_1, \ldots, CD_m\}$ (with $m \leq n$) that, after suitably assembled with the components in S, realize C*. Note that the number of delegates can be less than the number of CHOReOS components. For instance, as detailed in Section 4.2, a delegate is not needed for CHOReOS components that act as providers only (i.e., that externally expose only provided interfaces). Still, coordination delegates further serve implementing the mapping of base connector type roles to/from corresponding GA connector roles.

By referring to Chapter 2, when saying: "*Given a choreography specification C and a set of CHOReOS components $S = \{S_1, \ldots, S_n\}$, ...*" in the problem definition, we mean that we consider a set of LTS-based behavioral specifications, one LTS for each component in S. Informally, for coordination purposes, a state of the LTS models a "logical" state of the component. This logical state represents a certain point of the component interaction in which the component has performed an operation (i.e., one of the operations labeling the incoming transitions) and is ready to perform some other operation (the ones labeling the outgoing transitions). Note that, in this sense, the state of an LTS does not model the internal actual state of the component (e.g., values of its session attributes), rather its externally observable (logical) state.

Analogously, the choreography model that we have in mind for coordination purposes will also be LTS-based, where transitions are labeled with the service operation names (possibly extended in the future with I/O data plus non-functional annotations). The rationale behind this choice is that LTSs constitute a fundamental model of concurrent computation which is widely used in light of its flexibility and applicability. LTSs are often used as semantic model for many formal languages that are used to model concurrent systems. Example of these languages are CCS [50], and CSP [67]. Actually, very often, these calculi are formalized operationally by using an LTS-based semantics [80]. Furthermore, an LTS can be seen not only as a semantic model but also as a notation for behavioral specification purposes, which is also amenable of easy manipulation for automated analysis and synthesis [16, 18, 46, 49, 12, 60, 70, 72, 78, 35, 10].

In particular, for synthesis purposes, in CHOReOS we will define a model-to-model transformation (to be defined in WP2) to automatically derive the LTS-based specification of the choreography from a BPMN2 specification of it. To this end, it is worth to note that BPMN2 can be used for creating simple and rather abstract choreography specifications (that can be useful for providing, e.g., a business manager with a high-level view of a given business process), but also detailed and technical specifications (that can be parsed by a machine and automatically manipulated by developers and analysts, e.g., for tool-supported analysis and synthesis). Within CHOReOS, this peculiarity of BPMN2, creates the opportunity to align IT people and business people, as well as end-users, over the same goals and requirements, avoiding to provide business people and end-users with too many technical details, and IT people with inaccurate specifications.

In few words, the main novelties of our choreography synthesis approach with respect to the state of the art (see Deliverable D1.1 [75]) are: it efficiently produces a decentralized "choreographer" model without passing through a centralized one; it overcomes the ultra-large number of services by relying on a hierarchical service base (see Deliverable D2.1 [76]) where services are grouped into classes of functional and non-functional abstractions; and it is fully-automated. At this stage of the CHOReOS project, we do not consider non-functional properties whose treatment is left as future work.

In the next section, we characterize the main "ingredients" required to tackle the choreography-based coordination problem from an architectural point of view. That is, we give an overall description of the CHOReOS architectural style by putting its constituent entities all together, i.e., CHOReOS components,

CHOReOS connectors, and CHOReOS coordination delegates; and formally characterize the notion of FI choreography-based coordination.

4.2. Formal Abstractions for FI Choreography-based Coordination

As already introduced, the CHOReOS architectural style allows us to address the choreography-based coordination problem described above by using an automated synthesis approach. That is, on the practical side, the CHOReOS synthesis approach adopts a model-to-code transformation that, taking as input the specification of the choreography and the behavioral specifications of the participant components, produces as output the actual code that implements the coordination logic. For instance, the logic of a coordination delegate can be synthesized as an XML-based description that can then be translated into some executable form, as in particular supported by the CHOReOS Service-Oriented Middleware (see Deliverable D3.1 and embedded design of the XSC - eXecutable Service Composition)

The style defines the rules used to build the overall system and it is called CBA (i.e., *Choreography-Based Architecture*) style. Within this style, we consider a kind of systems where a set of coordination delegates is exploited to distributively realize the coordination logic implied by the desired choreography. More precisely, a CBA is made of a set of CHOReOS components where consumers and providers are composed with (e.g., wrapped by) their coordination delegates. Each delegate is connected to all the other delegates, through CHOReOS connectors, in an asynchronous way for additional communication. Furthermore, delegates can be connected to providers/consumers by CHOReOS connectors. Figure 4.1 illustrates an example of a CBA where:

- $S1$, $S2$, $S3$ and $S4$ are instances of CHOReOS components, with $S4$ being a provider only;
- $CD1$, $CD2$, $CD3$ are CHOReOS coordination delegates that distributively cooperate to realize the CHOReOS choreography;
- the communication channels denoted as lines between components are CHOReOS GA connectors.

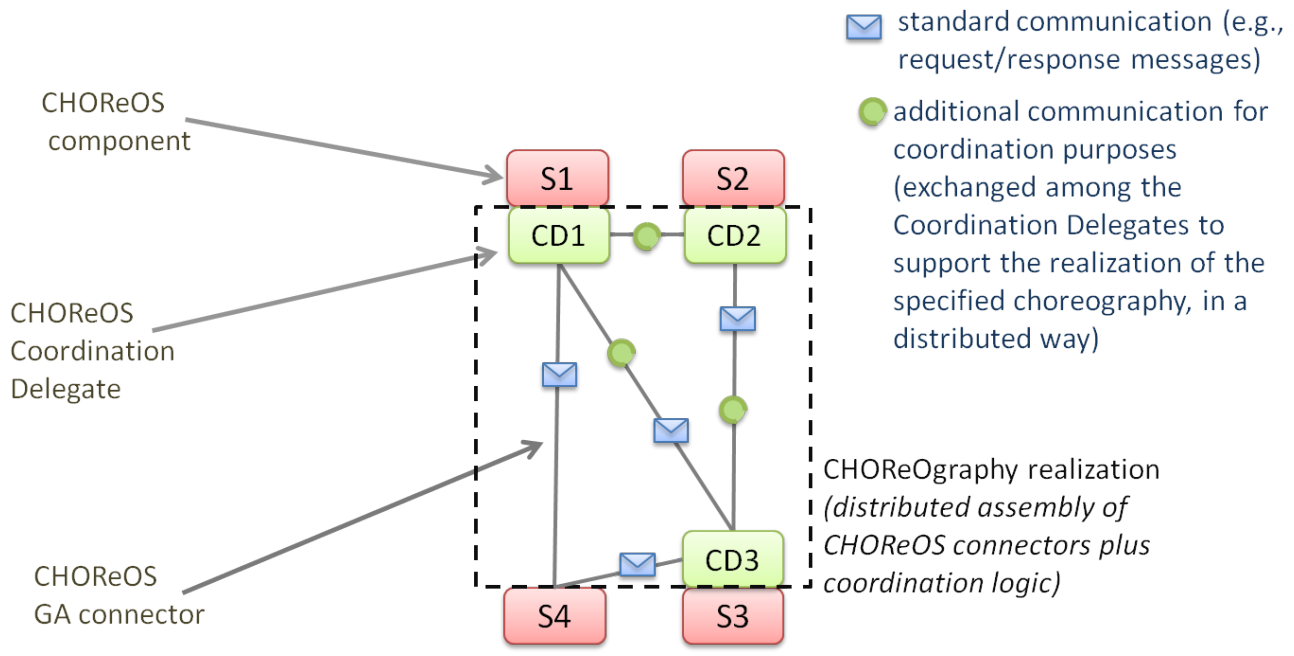


Figure 4.1: CBA: Architectural style for choreography-based coordination

Our choreography-based coordination style logically distinguishes between (i) *standard* and (ii) *additional communication*. The former denotes the operations that the components perform as described by their interface specification (which can include the component LTS-based behavioral specification, e.g., WSDL[1] plus WSCL[2], WSDL plus BPEL[3], etc.). The latter denotes additional information that the distributed coordination delegates *asynchronously* exchange in order to coordinate each other. Coordination delegates prevent undesired interactions (violating the specified choreography) by coordinating standard communication through the exchange of additional communication, **when needed**. More specifically, sharing some similarities with [35, 10], additional communication serves to keep track of the information about the "global state" of the coordination protocol (as implied by the specified choreography) that each delegate can deduce from the observed standard communication flow. As better explained later, additional communication is exchanged only when synchronization is needed, i.e., when there is more than one component that is allowed to perform some action according to the current global state of the choreography model. Note that this limits the overhead due to the exchange of additional communication to the strictly necessary minimum.

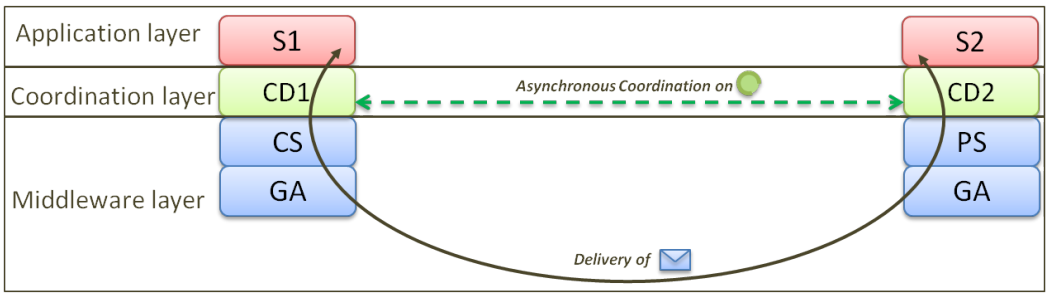


Figure 4.2: Sample architectural configuration: A two-layer view

The layered view in Figure 4.2 shows an example of a possible architectural configuration that, while conforming to the CHOReOS style, connects two components S_1 and S_2. The figure provides a two-layer point of view where the application and middleware layers are distinguished. In other words, it shows how interaction protocol coordination happens at application layer, and interaction paradigm mediation happens at middleware layer.

In the following, we provide the formal abstractions needed to characterize and reason on the entities of the CBA style, which are relevant for coordination purposes. The definitions we are going to provide hold for the initial version of the CHOReOS architectural style that we are describing in this deliverable. Then, at M24, we will finalize the architectural style definition and refine these definitions according to the technical work also from other RTD work packages, further accounting for the specific features of BMPN2 as our standard notation for choreography specification.

4.2.1. LTS-based behavioral specification of CHOReOS components

As introduced in Chapter 2, the interface of a CHOReOS component, among other things, can optionally expose a behavioral specification in the form of an LTS. In this section, we formally define this LTS-based specification that essentially describes the behavior of a component observable from outside and, hence, it is relevant for coordination purposes. We remark that the externally observable behavior of a component can be expressed in terms of sequences of operations belonging to the provided and required interfaces of the component.

Let Op be the universal set of component operations, and $Op_\tau = Op \cup \{\tau\}$, where τ denotes an internal operation that is not *observable* from outside.

[1] http://www.w3.org/TR/wsdl.
[2] http://www.w3.org/TR/wscl10/.
[3] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

An LTS L is a quadruple (S, T, D, s_0), where S is a finite set of states, $T \subseteq \{?, !\} \times Op$ is a set of transition labels (i.e., typed operation names) called the alphabet of L. $D \subseteq S \times (T \cup \{\tau\}) \times S$ is the transition relation and $s_0 \in S$ is the initial state. L is finite if D is finite and L is empty if D is empty. We will make use of the following notations: $g \xrightarrow{\alpha} h \iff (g, \alpha, h) \in D$; α=$?a$ (resp., α=$!a$) for some $a \in Op$ $\iff (?, a) \in T$ (resp., $(!, a) \in T$).

A typed operation name $?a$ (resp., $!a$) denotes a *provided* (resp., *required*) operation. An LTS $L = (S, T, D, s_0)$ is *non-deterministic* if $\exists (s, \alpha, s'), (s, \alpha, s'') \in D : s' \neq s''$, otherwise L is *deterministic*.

Intuitively, a component LTS behaves in a manner similar to that for finite state machines; it starts in the initial state and evolves over time according to its transition function. However, within the CBA style, we model component interaction by assuming that the components to be assembled communicate by means of synchronous communication channels. As it becomes clear hereafter, note that this is not a limitation. Thus, following the finite state case, a transition labeled by either a provided or a required operation may only be picked when the environment is respectively producing or consuming a request for that operation. In other words, both provided and required operations are considered to be blocking. That is, although enabled in the current state, they cannot be taken at any time; it depends on whether the environment is willing to synchronize on that operation or not.

Definition 15 (Trace) *Let $L = (S, T, D, s_0)$ be an LTS, $t = \mu_{i+1}\mu_{i+2} \dots \mu_n$ is a trace of L iff there exists a sequence of states $s_i, \dots, s_n \in S$ such that $s_i \xrightarrow{\mu_{i+1}} s_{i+1} \dots \xrightarrow{\mu_n} s_n$, $i \geq 0, n > i$. The empty trace is denoted by ϵ.*

The set of all traces of the LTS L starting from a state s_i is denoted as $Tr(L, s_i)$.

Definition 16 (Normalized trace) *Given a trace $t = \tau^* \mu_{i+1} \tau^* \dots \tau^* \mu_n \tau^*$ in $Tr(L, s_i)$, the normalized version of t is the trace $t^\tau = \mu_{i+1} \dots \mu_n$. Note that, the normalized version of τ^* is the empty trace ϵ.*

Given the set of traces $Tr(L, s_i)$ of an LTS L, we denote the corresponding *set of normalized traces* as $Tr(L, s_i)^\tau$. Thus, the externally observable behavior of a component can be defined as follows:

Definition 17 (Trace-based semantics of the component externally observable behavior) *The externally observable behavior of a component modeled by means of an LTS $C_i = (S, T, D, s_0)$ is:*

$$Tr(C_i, s_0)^\tau$$

LTSs can be used to define finite state systems [73]. For our purposes, we assume that all systems we deal with are finite state systems. Note that, in our context, this is not a restriction. In fact, each component exports through its interface a finite number of "operations". In our model, each operation of a component interface can be seen as a point of interaction of the component with the other components. If we modeled all the possible externally observable component interactions with a "classical" automaton instead of an LTS, we would also have accepting states. Indeed, for our purposes, what matters about a particular component interaction is not whether it drives the automaton in an accepting state but whether the automaton is able to perform the corresponding sequence of actions interactively. Thus, we should consider an automaton in which *every* state is an accepting state [50] (i.e., an LTS). A consequence is that if an automaton accepts a particular component interaction seen as a sequence of component interface operation invocations, then it also accepts any initial part of that interaction/sequence. In other words, due to the finiteness of the set of component interface operations, although all the possible component interactions can be infinite we can always finitely represent them since the language built over the component interface operations (i.e., the model of the component interaction behavior) is prefix-closed [36]. Prefix-closed languages are generated by prefix-grammars that describe exactly all regular languages.

Following this discussion, and to precisely understand the remainder of the chapter, we have to consider that communication channels can be:

- *asynchronous* - no synchronization points exist and the interaction never blocks the sender. This implies a potentially unbounded buffer; in practice, a bounded buffer is used and the sender will block when the buffer is full. In this way, a higher degree of parallelism can be achieved because (possibly) the sender never has to wait. For instance, in a message-passing setting, when a peer executes a send action, the sent message is added to the tail of the receive buffer (i.e., the queue) of the receiver, and a peer can consume the message at the head of its receive queue by executing a receive action.
- *synchronous* - no buffer is used and, due to synchronization points, both senders and receivers can block. The term rendezvous is often used to evoke the image of two processes that have to meet at a specific synchronization point. For instance, in a message-passing setting, sender and receiver move in lock-step, i.e., a send action of a sender is allowed only when the receiver is ready to perform the corresponding receive action. The sender is blocked until the message exchange ends. It happens when the sender receives a notification from the receiver.

As already introduced, we model component interaction by assuming that the components to be assembled communicate by means of synchronous communication channels. This is not a limitation since, in practice, by introducing a finite buffer component (i.e., a finite queue) to decouple the interaction, we can simulate a bounded asynchronous system with a synchronous one [79, 50]. Obviously, in this case, there is the necessity of explicitly programming the needed buffers by exploiting the native primitives that are provided to support the synchronous communication.

It is well known that reasoning with the presence of unbounded buffers can lead to undecidable verification problems [14]. Moreover, strictly concerning the CHOReOS purposes, the choreography conformance problem of identifying if a set of interacting services can fulfill a desired choreography specification is, in general, undecidable when unbounded asynchronous communication is used [14, 10]. From a practical point of view, this motivates the reasonable restriction to consider only synchronous systems or bounded asynchronous ones that can always be modeled as finite state machines by assuming synchronous communication.

Indeed, in [10], the authors show how for a particular class of systems, where finite state peers are assumed to communicate with *unbounded* message queues, the choreography conformance problem can be solved. This is done by comparing the behavior of the peers with synchronous communication to the one with bounded asynchronous communication where each message queue is restricted to a queue of size 1 (i.e., if there is already a message in the queue, then the send actions that try to send to that queue block). From a theoretical point of view this approach constitutes a valuable result.

4.2.2. LTS-based specification of the choreography

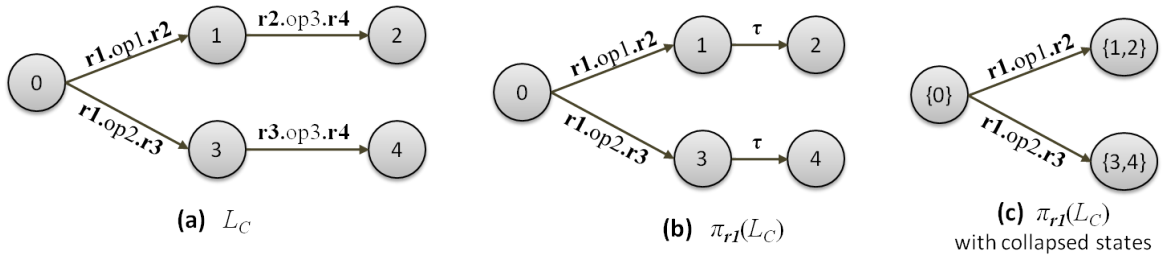
As introduced above, by means of a model-to-model transformation to be defined in WP2, we derive an LTS-based specification of the choreography from a BPMN2 specification of it. In this section, we formally define the LTSs we plan to adopt for the initial version of the CHOReOS architectural style defined in this deliverable.

Definition 18 (Choreography LTS) *A Choreography LTS L_C is pair $(l_c, Roles)$. $Roles$ is the universal set of participant roles, and l_c is an LTS (S, T, D, s_0) where $T \subseteq Roles \times Op \times Roles$ is the set of transition labels. An element of T is a triplet (r, α, r') representing an operation α required by r and provided by r'. We denote it as $r.\alpha.r'$.*

The notions of non-determinism and trace-based semantics are straightforward from Definitions 14 and 17.

As also mentioned in Deliverable D2.1 [76], the interaction flow globally defined by the choreography can be projected, in a peer-style fashion, among different participants according to the "local" roles that they play to fulfill the global choreography, hence obtaining a *peer-style specification* of the

choreography. Note that, the notions of role and operation abstract the notions of participant and task, respectively, of a BPMN2 choreography specification.



(a) L_C (b) $\pi_{r1}(L_C)$ (c) $\pi_{r1}(L_C)$ with collapsed states

Figure 4.3: A choreography LTS (a), its projections onto $r1$ ((b), (c)-with collapsed states)

In other words, a Choreography LTS L_C can be projected to one of its roles, denoted as $\pi_r(L_C)$, which is obtained by replacing every transition where r is neither the consumer nor the provider by a τ-transition. States linked by τ-transitions can be collapsed to a single state whose label denotes the set of collapsed states. Figure 4.3.(a) shows a Choreography LTS L_C, Figure 4.3.(b) shows its projection $\pi_{r1}(L_C)$, and Figure 4.3.(c) shows $\pi_{r1}(L_C)$ with collapsed states.

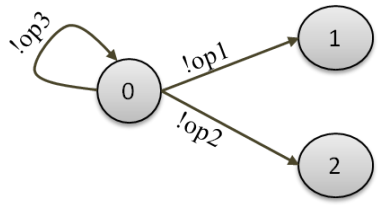


Figure 4.4: Projection refinement sample

By referring to the third scenario previously introduced, it is worth to mention that the peer-style specification can be useful to discover[4] those services whose behavior can be potentially suitable to fulfill the choreography. Roughly speaking, a suitable notion of *refinement* can be exploited to check whether the LTS-based behavioral specification of a discovered service refines the choreography projection onto the corresponding role. It is well known that refinement checks over LTSs can be implemented by means of *trace containment* [50]. That is, an LTS $L_1 = (S_1, T_1, D_1, s_0^1)$ refines another LTS $L_2 = (S_2, T_2, D_2, s_0^2)$ if $Tr(L_2, s_0^2)^\tau \subseteq Tr(L_1, s_0^1)^\tau$. In our case, L_1 is the LTS-based specification of a component and L_2 is a Choreography LTS projection. Clearly, while performing trace containment, we have to take into account that two syntactically different labels can have the same semantics, e.g., α on L_1 and $r.\alpha.r'$ on L_2. For instance, the LTS-based specification shown in Figure 4.4 refines the projection $\pi_{r1}(L_C)$. This means that the component modeled by the LTS in Figure 4.4, if suitably coordinated, can play the role $r1$.

4.2.3. CHOReOS coordination protocol

This section formally characterizes the notion of CHOReOS coordination protocol, which is implied by both a Choreography LTS L_C and the LTS-based behavioral specifications, $L_1, \ldots, L_n$, of the participant components. To this end, we recall that the additional communication among coordination delegates serves to keep track of the global state of the coordination protocol implied by L_C that each delegate can deduce from the observed standard communication flow among $L_1, \ldots, L_n$.

The goal is to distribute L_C in a way that each coordination delegate knows which operation the supervised component is *allowed* to execute. We refer to these operations as *allowed operations*. Al-

[4] Service discovery is worked out in WP2 and WP3.

lowed operations are used by the coordination delegates as basis for correctly synchronizing with each other by exchanging additional communication. In other words, the delegates interact with each other to restrict the components' standard communication by allowing only the part of the communication that is correct with respect to L_C. In the following, we formally characterize the coordination information concerning allowed operations which is needed for a delegate to correctly exchange additional communication.

Let L_C=$((S_C, T_C, D_C, s_0), \{r_1, \ldots, r_n\})$ be the Choreography LTS. Let $1, \ldots, n$ be the unique identifiers for n participant components playing the roles $r_1, \ldots, r_n$, respectively. Let C_1=(S_1, T_1, D_1, s_0^1),...,$C_n = (S_n, T_n, D_n, s_0^n)$ be their LTS-based behavioral specifications. Then, the *coordination information* $Coord_i$, for the required operations of component i, is formally defined as follows:

$$Coord_i=\{\langle s, !l, s', AC_s, AC_{s'}\rangle \mid \exists (!, l) \in T_i : (s, r_i.l.r_x, s') \in D_C \textit{ for some } r_x \in \{r_1, \ldots, r_n\}\}$$

where

$$AC_s=\{j \neq i \mid \exists (!, l') \in T_j : (s, r_j.l'.r_x, s'') \in D_C \textit{ for some } s'' \in S_C, \textit{ and some } r_x \in \{r_1, \ldots, r_n\}\}$$

$$AC_{s'}=\{k \neq i \mid \exists (!, l'') \in T_k : (s', r_k.l''.r_x, s''') \in D_C \textit{ for some } s''' \in S_C, \textit{ and some } r_x \in \{r_1, \ldots, r_n\}\} \bigcup \{k \neq i \mid \exists (?, l'') \in T_k : (s', r_x.l''.r_k, s''') \in D_C \textit{ for some } s''' \in S_C, \textit{ and some } r_x \in \{r_1, \ldots, r_n\}\}$$

According to the transitions D_C of the choreography L_C, the first three elements of each quintuplet in $Coord_i$ represent the allowed operations that can be *required* by the component C_i (playing the role r_i) while the global state of L_C is s. For an allowed operation $!l$, the fourth element AC_s is the set of (identifiers of) other *active components* in s, i.e., the ones that can also require some (possibly different) allowed operation from s. Similarly, after that the operation $!l$ has been performed by C_i, the fifth element $AC_{s'}$ is the set of active components in s', i.e., the ones that can require or provide some allowed operation from s'.

In particular, once L_C has been distributed into the various $Coord_i$, each delegate knows the states of L_C from which it can allow the supervised component to perform a specific operation. Moreover, once a component i is allowed to perform such an operation, e.g., labeling a transition from a state s to a state s' of L_C, its delegate knows also which are the components that must be *blocked* (i.e., the ones in AC_s) before performing the operation, and which ones must be *unblocked* (i.e., the ones in $AC_{s'}$) after the operation has been performed. Let us assume that a component C_i is going to perform an operation contained in $Coord_i$. If it can proceed according to the current state s of L_C, then all the other active components in s are blocked by sending a blocking message to the corresponding delegates; this is needed because otherwise, e.g., two different components could simultaneously proceed in two different states of L_C hence leading to a global state that is inconsistent with respect to L_C. Once C_i has performed the action, all the components that can move in the new state s' of L_C are unblocked[5] and they are made aware of the new current state s' of L_C; this is needed because some components that now can move could have been previously blocked. It is worth to note that, from the current state s, it is sufficient to block components only upon required operations. Components upon provided operations do not need to be blocked because they are already waiting for the operation request. This explains the above definition of AC_s. Contrarily, once the new current state s' has been reached, components have to be unblocked upon both required and provided operations because, e.g., a component, that in s' provides an allowed operation, might have been blocked in s upon some other required operation. This explains the above definition of $AC_{s'}$.

Blocking and unblocking messages (together with acknowledging messages) concern the additional communication that is asynchronously exchanged among delegates for coordination purposes. It is

[5] By their supervising coordination delegates that have received an unblocking message by the delegate supervising C_i.

worth to mention that the distribution of L_C into the various $Coord_i$ of the various delegates, one for each component, can be efficiently obtained by means of a depth-first visit of L_C and, hence, its computational complexity is polynomial in the number of states of L_C. This computational complexity reveals that the elicitation of the distributed coordination logic out of the Choreography LTS can be performed for large-scale contexts (with respect to the size of the Choreography LTS).

Before introducing the distributed coordination algorithm performed by the coordination delegates (see Section 4.2.4), in the remainder of this section, we give some preliminaries on *happened-before* relation and *partial ordering* [43] that might be needed for the full understanding of such an algorithm. The reader that is familiar with this notion may want to skip this part.

Happened-before relation and partial ordering: We now briefly recall basic notions related to distributed systems and recall the well known *time-stamp method* that we use within the distributed coordination algorithm described in Section 4.2.4.

Without loss of generality, we assume that the components to be choreographed are uniquely identified and assigned to different processors (residing in different interconnected machines). Note that, this assumption can be maintained to model abstract parallelism when more than one component is assigned to a single processor. For the sake of clarity, we also assume that each component is single-threaded and, hence, all its send and receive events can be totally ordered to constitute a set of traces. Note that, this is not a restriction since a multi-threaded component can always be modeled as a set of single-threaded (sub-)components simultaneously executed.

Considering synchronization on common required/provided operations, the corresponding send and receive events are considered to occur simultaneously. Interaction among components is then modeled by *interleaving* of traces. This means merging two or more traces such that the independent events (i.e., not common) from different traces may occur in any order in the resulting trace, while the events within the same traces retain their order. A trace resulting from this merge is usually called a *linearization* [79].

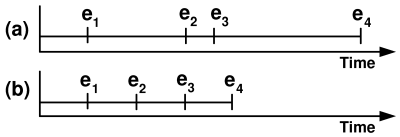


Figure 4.5: Time unit and sequence of events

In a concurrent and distributed context, we cannot assume either a single physical clock or a set of perfectly synchronized ones in order to determine whether an event a occurs before an event b or vice versa. We then need to define a relationship among the system events by abstracting from both the absolute speed of each processor and the absolute time. In this way we ignore any absolute time scale and assume each event to be executed in a time unit[6]. For example, the sequence of events depicted in Figure 4.5.(a) can have different execution times based on different processor, but we will model it as in Figure 4.5.(b) where each event occurs in a prefixed time unit.

By taking into account the *law of causality* [43] (e.g., a message can be received only after it has been sent), it is possible to define the *happened-before* relation (denoted by $\rightarrow$) on a set of events as follows:

Definition 19 *(**Happened-before relation**). Let E_i be the set of events of a component C_i and let $E = \bigcup_{i=1}^{n} E_i$ be the set of all possible system events, then the happened-before relation $\rightarrow \subseteq E \times E$ on the set of events of a system is the smallest relation satisfying the following three conditions:*

1) if e_1 and e_2 are events of the same component and e_1 is executed before e_2, then $e_1 \rightarrow e_2$;

[6] This abstraction is acceptable since we are not interested in real-time systems for which the absolute time is relevant.

2) *if* e_1 *is the event corresponding to the sending of a message* m *by a component and* e_2 *is the event corresponding to the receiving of* m *by another component, then* $e_1 \rightarrow e_2$*;*

3) *if* $e_1 \rightarrow e_2$ *and* $e_2 \rightarrow e_3$ *then* $e_1 \rightarrow e_3$*.*

For each pair of distinct events $e_1, e_2 \in E$, if e_1 does not occur before e_2 and e_2 does not occur before e_1 we will write $e_1 \nrightarrow e_2$ and $e_2 \nrightarrow e_1$. In this case we say that a and b are concurrently executed. Obviously, $e_i \nrightarrow e_i$ for each $e_i \in E$ since e_i cannot occurs before itself.

We refer to [43] for a detailed discussion about the concepts of concurrency and happened-before relation. Note that, the happened-before relation is only a partial ordering of the events in distributed systems. In particular, when modeling synchronous communication, we can consider, e.g., send and receive events of the same message to occur simultaneously. Thus, in this case, we might restrict the partial order to only send-events.

Timestamps and total ordering: Within the algorithm described in Section 4.2.4, we use the well known *time-stamp method* [43] to implement the happened-before relation. Whereas the events onto a same component can be totally ordered, the relation with the events of different components is not always well defined. By the time-stamp method it is possible to define a global order among the whole events (send/receive) exchanged through the choreography-based system for each component. In other words, each component can establish a total order among its generated events and the events received from other components.

The idea is to associate a time-stamp to each event. This is just a number that each sending component associates to its messages. Locally, time-stamps are sequentially generated, one for each event. Whenever a receive event e' occurs at a component S_i, such a component is able to determine a local order among its own events and e'. For instance, if S_i intended to send a message e with a time-stamp lower than the one associated with e', then e is processed before e', i.e., $e \rightarrow e'$. Moreover, in order to try to synchronize with the sending component, S_i will use the received time-stamp plus one as next time-stamp, i.e., the next message that S_i wants to send will be associated with the updated time-stamp. On the other hand, if the time-stamp associated to e is bigger than the received one, S_i considers $e' \rightarrow e$. Finally, if the time-stamp associated with e is equal to the time-stamp associated with e', then there is concurrency. In this case, in order to avoid any ambiguity, an order relation $\prec$ among components can be a priori fixed. If e' was sent by a component S_j such that $S_j \prec S_i$ (i.e., $j < i$) in the fixed order, than $e' \rightarrow e$, otherwise $e \rightarrow e'$, hence obtaining a total order among events. As described in [43], "fairer" methods can be used. Note that the time-stamp method is a lightweight and easy way to validate mutual exclusion, freedom from deadlock and starvation, and it allows one to state other useful and crucial properties for distributed systems (e.g., minimum overhead in performing synchronizing communication).

4.2.4. Distributed coordination algorithm

While exchanging additional communication, the standard time-stamp method is used in our approach in order to avoid problems of synchronization. In this way an ordering among dependent blocking and unblocking messages is established and starvation problems are addressed. Such a method establishes a total order at each coordination delegate among the sent and received blocking and unblocking messages. We recall that for this purpose we also need an ordering among components. Such an *a priori* fixed order (e.g., based on $\prec$ as mentioned above) solves concurrency problems arising when two events with the same associated time-stamp must be compared.

Adopting the same presentation style as the one used in [43], our distributed coordination algorithm is defined by the following rules that each coordination delegate CD_i independently follows, when S_i

performs a request of α, without relying on any central synchronizing entity or shared memory. Roughly speaking, these rules locally characterize the collaborative behavior of the coordination delegates at run-time from a clear "*one-to-many*" point of view. Note that this view can be useful for model-to-code transformation since it precisely defines the (parametric) coordination logic of a generic coordination delegate CD_i.

Each coordination delegate maintains its own *BLOCK queue* (i.e., the queue of blocking messages) that is unknown to the others delegates. At the beginning, each coordination delegate has its own timestamp variable set to 0 and, at each iteration of the algorithm, waits for either its supervised component to make a request or another coordination delegate to forward a request. The algorithm is then defined by the following rules. The actions defined by each rule has to be considered as atomic. Within the rules, we denote with TS_i the current timestamp for CD_i, and with s the current state of L_C. Furthermore, $Coord_i[h]$ is the h-th tuple contained in $Coord_i$ and $Coord_i[h][j]$ is its j-th element.

Rule 1: Upon receiving , from S_i, a request of α in the current state s of L_C,

- **1.1** if there exist h ***s.t.*** $Coord_i[h][1] = s$ and $Coord_i[h][2] = \alpha$ (i.e., α is allowed from s),
  - **1.1.1** for every CD_j ***s.t.*** $j \in Coord_i[h][4]$,
    - **1.1.1.1** CD_i updates TS_i to $TS_i + 1$;
    - **1.1.1.2** CD_i sends BLOCK(s,TS_i,CD_i) to CD_j;
    - **1.1.1.3** CD_i puts $\langle$BLOCK$(s, TS_i, CD_i), CD_j\rangle$ on its BLOCK queue;
- **1.2** if there exist h ***s.t.*** $Coord_i[h][1] \neq s$ and $Coord_i[h][2] = \alpha$ (i.e., α is not allowed from s), CD_i discards α;
- **1.3** if does not exist h ***s.t.*** $Coord_i[h][2] = \alpha$ (i.e., α is not in the alphabet of L_C), CD_i forwards α (hence synchronizing with S_i).

Rule 2: When a CD_j receives a BLOCK(s,TS_i,CD_i) from some CD_i with its own TS_i,

- **2.1** CD_j places $\langle$BLOCK$(s, TS_i, CD_i), CD_j\rangle$ on its BLOCK queue;
- **2.2** if $(TS_j < TS_i)$ or $(TS_i = TS_j$ and $S_i \prec S_j)$ then CD_j updates TS_j to $TS_i + 1$, else CD_j updates TS_j to $TS_j + 1$;
- **2.3** CD_j sends[7] ACK(s,TS_j,CD_j) to CD_i.

Rule 3: Once CD_i has received all the expected ACK(s,TS_j,CD_j) from every CD_j (see Rule 2), and it is granted the privilege (according to Rule 5) to proceed from state s,

- **3.1** CD_i forwards α;
- **3.2** CD_i updates s to s'=$Coord_i[h][3]$;
- **3.2** for every CD_j ***s.t.*** $j \in Coord_i[h][5]$ (i.e., to unblock the components allowed to perform some operation from the next state s' of L_C),
  - **3.2.1** CD_i removes $\langle$BLOCK$(s, TS_i, CD_i), CD_j\rangle$ from its own BLOCK queue;
  - **3.2.1** CD_i updates TS_i to $TS_i + 1$;
  - **3.2.2** CD_i sends UNBLOCK(s',TS_i,CD_i) to CD_j.

Rule 4: When a CD_j receives an UNBLOCK(s',TS_i,CD_i) from some CD_i with its own TS_i,

- **4.1** CD_j updates s to s';
- **4.2** if $(TS_j < TS_i)$ or $(TS_i = TS_j$ and $S_i \prec S_j)$ then CD_j updates TS_j to $TS_i + 1$, else CD_j updates TS_j to $TS_j + 1$;

[7] There is no need to send this ACK if CD_j has already sent a message (BLOCK or UNBLOCK) to CD_i with a timestamp later than TS_i.

4.3 CD_j removes any $\langle$BLOCK$(s, TS_i, CD_i), CD_j\rangle$ from its BLOCK queue[8];

4.4 CD_j retries Rule 1 from the updated state s.

Rule 5: CD_i is granted the privilege to proceed from the current state s of L_C when the following condition is satisfied: there is a $\langle$BLOCK$(s, TS_i, CD_i), CD_j\rangle$ message on its BLOCK queue *s.t.* for every other $\langle$BLOCK$(s, TS_j, CD_j), CD_i\rangle$ message on its BLOCK queue either (i) $TS_i < TS_j$ or (ii) $TS_i = TS_j$ and $S_i \prec S_j$.

Rule 6: Upon receiving , from some CD_j, a request of α, CD_i forwards α to S_i.

For a full understanding of the algorithm, in the following, we provide a detailed explanation of some rules.

If the conditions on Rule 1.2 hold (i.e., the conditions on Rules 1.1 and 1.3 fail), it means that the component supervised by CD_i is trying to perform an operation that is in the alphabet of L_C but is not allowed from the current state of L_C. In this case, CD_i prevents S_i to perform that operation by discarding it. Indeed, one cannot always assume that the actual code of a (black-box) component has been developed in a way that it is possible to discard a component operation by the external environment. Actually it can be done only if the developer had preemptively foreseen it and, for instance, an exception handling logic was aptly coded for such an operation[9]. Thus, we would need to distinguish between *controllable* and *uncontrollable* actions. In other words, we should distinguish between component operation that can be discarded by the external environment (e.g., a coordination delegate) and component operations that cannot be discarded. For example, inputs coming from a sensor are often considered as uncontrollable since they must be accepted and treated by the component. In contrast, controllable actions can be safely discarded to correctly prevent undesired behaviors. As it is usually done in the *discrete controller synthesis* research area [64, 15], the developer is in charge of specifying which component operations are controllable and which are uncontrollable. Therefore, for the purposes of our method, we should assume that the component developer specifies this kind of information, e.g., by tagging, within a component LTS, operation labels as controllable or uncontrollable. Since in this deliverable we mainly focus on the automatic distribution of the choreography-based coordination logic, for the sake of simplicity, we avoid to address controllability issues and we assume that all component operations are controllable. This is not a limitation of the work presented in this deliverable since its extension to account for controllability issues is straightforward.

Rule 1.3, allows coordination delegates to be *permissive* on the operations that do not belong to the alphabet of L_C (i.e., operations "outside the scope" of the choreography). Note that one could instead choose to be *restrictive* on that operations by disabling Rule 1.3, hence preventing the component to perform that operation by discarding it (as in the case of a component trying to perform an operation that is in the alphabet of L_C but is not allowed from the current state). Clearly, choosing between either the permissive or the restrictive version of Rule 1.3 can be handled by considering coordination delegates parametric with respect to that rule (this parameter can be set at assembly time).

Rule 4.4 resumes the execution of an unblocked coordination delegate by "restarting" from Rule 1. If this coordination delegate is still trying to handle a request α that is *pending* from the previous iteration of the algorithm (see the operation `flightInfo2` in the coordination scenario shown in Figure 4.8), retrying Rule 1 means to directly re-check the conditions of Rules 1.1, 1.2, and 1.3 with the new updated state and the pending α. Otherwise, it means that the coordination delegate restarts from Rule 1 by waiting for either a new request from its supervised component or from another coordination

[8] For some execution, if each coordination delegate does not strictly alternate BLOCK and UNBLOCK messages, then executing Rule 4 may result in removing zero BLOCK messages.

[9] E.g., through declaration of *thrown* exceptions on interface operations, or of *fault messages* on WSDL operations, or simply of *error return values* for class methods.

delegate (Rule 6).

It is worthwhile noticing that conditions (i) and (ii) of Rule 5 are tested locally by a coordination delegate.

Correctness of the algorithm: It is easy to verify that the algorithm defined by these rules satisfies the three crucial conditions [43] for *correct* distributed coordination: (1) a coordination delegate which has been granted the privilege to proceed must proceed and unblock the other competing delegates before the privilege to proceed can be granted to another delegate; (2) different block messages for granting the privilege to proceed must be privileged in the order in which they are made, excluding the ones "associated" to discarded operations; (3) if every coordination delegate which is granted the privilege to proceed eventually proceeds and unblocks the other competing delegates, then every block message for granting the privilege to proceed is eventually privileged, excluding the ones "associated" to discarded operations. First of all, condition (i) of Rule 5, together with the assumption that the messages concerning additional communication are received in order, guarantees that CD_i has learned about all operation requests which preceded its current operation request. Since Rules 3 and 4 are the only ones which remove messages from the BLOCK queue, condition (1) trivially holds. Condition (2) follows from the fact that the total ordering $\prec$ (happened-before relation plus component priority) extends the partial ordering $\rightarrow$ (happened-before relation). Rule 2 guarantees that after CD_i requests the privilege to proceed (by sending BLOCK messages), condition (i) of Rule 5 will eventually hold. Rules 3 and 4 imply that if each coordination delegate which is granted the privilege to proceed eventually proceeds and unblocks the other competing delegates, then condition (ii) of Rule 5 will eventually hold, thus ensuring condition (3).

Analysis of the overhead due to the exchange of additional communication: It is very easy to see that the overhead due to the exchange of additional communication among the coordination delegates is negligible. First of all, note that BLOCK messages are exchanged only when non-determinism occurs from the current state s of L_C. In the worst case[10], the non-determinism degree is asymptotically bounded by the number n of components, i.e., it is $O(n)$. For each received BLOCK message an ACK message is exchanged. UNBLOCK messages are instead exchanged at each state of L_C and for a maximum number that is $O(n)$. Thus, if m is the number of states of L_C then the maximum number of additional communication messages (BLOCK, UNBLOCK, ACK) that are exchanged is $O(3 * m * n)$, i.e., $O(m * n)$. However, very often, in the practice, $n \leq m$ holds ($m \leq n$ is less frequent). This means that the maximum number of exchanged additional communication messages can be considered as $O(m^2)$. We can, then, conclude that the introduced overhead is polynomial in the number of states of L_C and, hence, negligible further considering that the size of additional communication messages is insignificant. After all, as also shown by the work described in [43], this is the strictly necessary minimum that one can do to ensure correct distributed coordination.

Illustrative example: Now, by means of a small example, we better show how coordination delegates use, at run-time, the information in $Coord_i$ to correctly and distributively interact with each other, hence ensuring the behavior globally specified by L_C. We consider the development of a choreography-based travel agency system that can be realized by choreographing four services: a `Booking Agency` service, two `Flight Booking` services, and a `Hotel Booking` service. As shown in Figure 4.6, starting from their initial states 0, `Booking Agency` makes requests to `Flight Booking 1`, `Flight Booking 2`, and `Hotel Booking` in order to book a flight and a hotel. The agency search for a flight

[10] Note that, in the practice, the worst case is unusual.

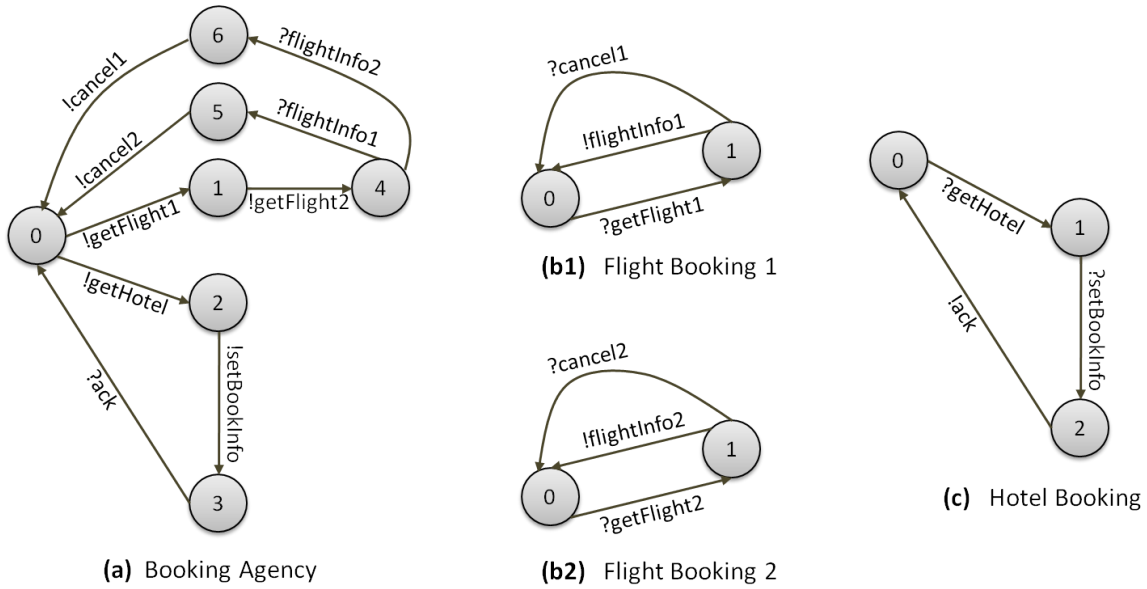


Figure 4.6: LTSs for travel agency services

by exploiting two different flight booking services. As soon as one of the two booking services answers by sending flight information (i.e., `!flightInfo1` or `!flightInfo2`), the agency cancels the search on the other booking service (i.e., `!cancel1` or `!cancel2`).

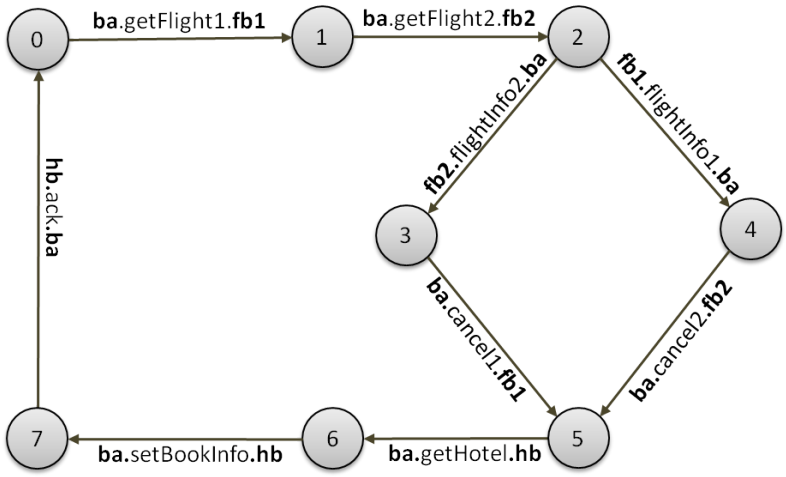


Figure 4.7: L_{FH}: Choreography LTS for a Flight-Hotel Booking collaboration

The Choreography LTS L_{FH}, shown in Figure 4.7, specifies that (i) flight booking has to be performed before hotel booking and (ii) only the answer from one of the two flight booking services is taken into account. Focusing on (ii) and following the rules of the distributed coordination algorithm, Figure 4.8 shows how `Flight Booking 2`, playing the role `fb2`, is blocked whenever `Flight Booking 1`, playing the role `fb1`, is faster in collecting the information to be provided to `Booking Agency`, playing the role `ba`.

The shown scenario concerns an excerpt of a possible execution of the distributed coordination algorithm. It starts when the two allowed operations `flightInfo1` and `flightInfo2`, required by `Flight Booking 1` and `Flight Booking 2` respectively, concurrently occur while in the current state 2 of L_{FH}. At state 2, the timestamps for `Flight Booking 1` and `Flight Booking 2` are 1 and 2, respectively. Furthermore, `Flight Booking 1` $\prec$ `Flight Booking 2`.

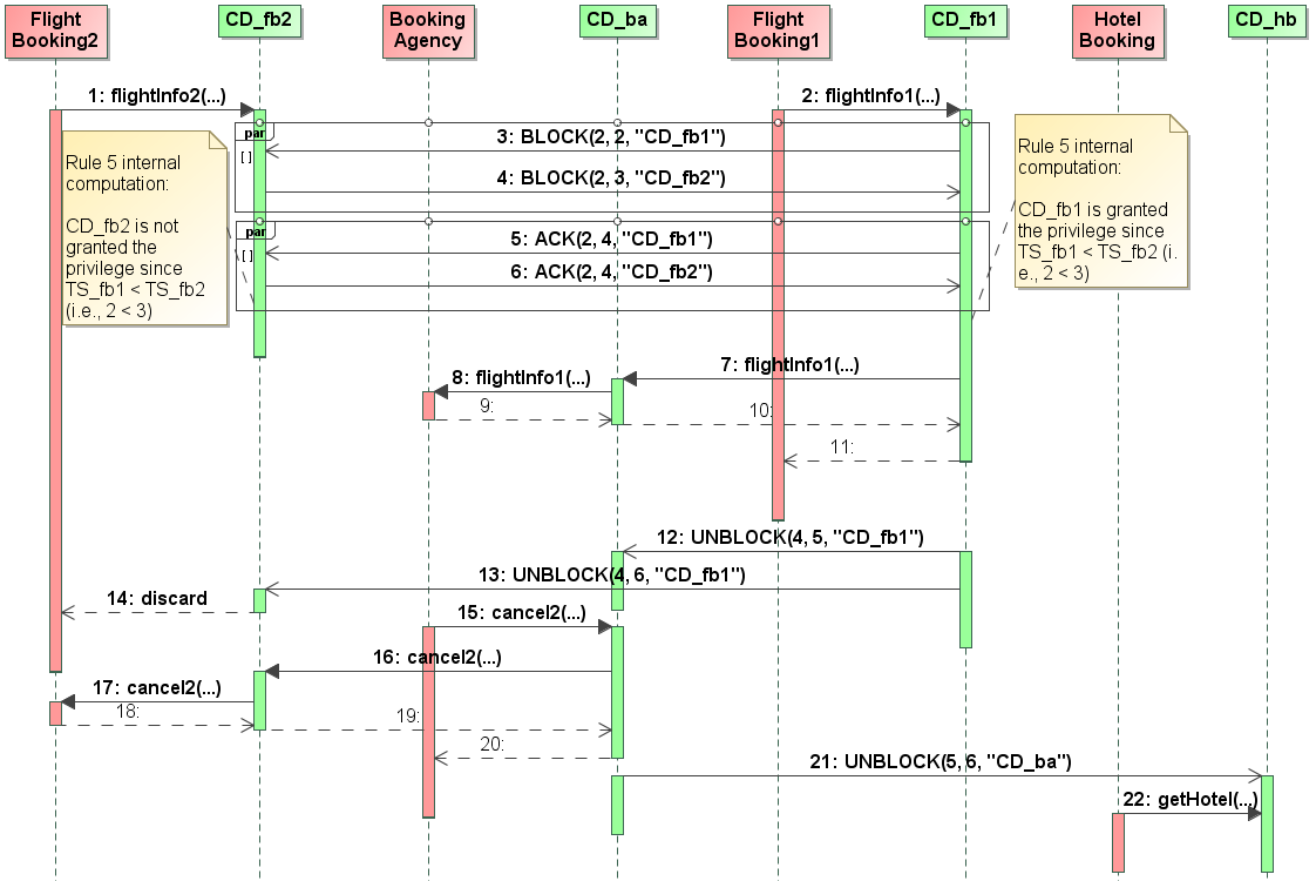


Figure 4.8: An excerpt of a possible execution of the distributed coordination algorithm

5 Conclusions and Future Work

This document has introduced the first version of the CHOReOS architecture style, which has informed the development of the CHOReOS IDRE and especially the work that has been undertaken within WP2 and WP3. Specifically:

- The CHOReOS service abstraction that is introduced in Chapter 2 shall enable sustaining the ultra large number of services aggregated in the FI, as well as their heterogeneity, in particular considering Business and Thing-based services that get inter-connected. The CHOReOS service abstraction specifically serves designing the abstraction-oriented service base within WP2 (see D2.1), which is integrated in the eXtensible discovery service of the CHOReOS SOM developed in WP3 (see D3.1).
- The GA connector type that is defined in Chapter 3 enables revisiting the bus paradigm to enable cross-paradigm interaction in the CHOReOS SOM and in particular interoperability across the Business and Thing domains, as reported in Deliverable D3.1.
- The formal abstractions for FI choreography-based coordination elaborated in Chapter 4 paves the way for the automated synthesis of concrete distributed coordination protocols, from BPMN2 abstract specification and associated concrete services discovered in the environment.

As a result, the proposed architectural style revisits the service oriented architecture paradigms to cope with the FI challenges, i.e., scalability, heterogeneity, mobility, and awareness & adaptability. However, this still needs to be assessed, which will be carried out based on the outcomes of RTD WPs, WP2 to WP5, possibly leading to make evolve the proposed style. In addition, as discussed along the chapters and in particular in Chapter 3 on the definition of connector types, some pending issues have already been identified among which accounting for continuous interactions, while this document has concentrated on discrete ones.

Bibliography

[1] E. Al-Masri and Q. H. Mahmoud. Discovering Web Services in Search Engines. *IEEE Internet Computing*, 12(3):74–77, 2008.

[2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3), 1997.

[3] D. Ardagna and B. Pernici. Adaptive Service Composition in Flexible Processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, 2007.

[4] D. Athanasopoulos and A. Zarras. Fine-grained Metrics of Cohesion Lack for Service Interfaces. In *Proceedings of the 9th International Conference on Web Services (ICWS)*, 2011.

[5] D. Athanasopoulos, A. Zarras, and V. Issarny. Service Substitution Revisited. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009. (short paper).

[6] D. Athanasopoulos, A. V. Zarras, P. Vassiliadis, and V. Issarny. Mining service abstractions: NIER track. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE)*, pages 944–947, 2011.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[8] E. Avilés-López and J.A. García-Macías. TinySOA: a Service-oriented Architecture for Wireless Sensor Networks. *Service Oriented Computing and Applications*, 3(2):99–108, 2009.

[9] A. Bakshi, A. Pathak, and V.K. Prasanna. System-level Support for Macroprogramming of Networked Sensing Applications. In *Int. Conf. on Pervasive Systems and Computing (PSC)*. Citeseer, 2005.

[10] Samik Basu and Tevfik Bultan. Choreography conformance via synchronizability. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 795–804, 2011.

[11] Françoise Baude, Imen Filali, Fabrice Huet, Virginie Legrand, Elton Mathias, Philippe Merle, Cristian Ruz, Reto Krummenacher, Elena Simperl, Christophe Hammerling, and Jean-Pierre Lorre. ESB Federation for Large-scale SOA. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2459–2466, New York, NY, USA, 2010. ACM.

[12] Sonia Ben Mokhtar, Nikolaos Georgantas, and Valérie Issarny. Cocoa: Conversation-based service composition in pervasive computing environments with qos support. *J. Syst. Softw.*, 80:1941–1955, December 2007.

[13] G.S. Blair, A. Bennaceur, G. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems. In *Proceedings of Middleware'2011 - To appear*, 2011.

[14] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30:323–342, 1983.

[15] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39, February 1994.

[16] Antonio Brogi and Razvan Popescu. Automated generation of bpel adapters. In *In Proc. of ICSOC'06, volume 4294 of LNCS*, pages 27–39. Springer, 2006.

[17] Cabri, G. and Leonardi, L. and Zambonelli, F. Reactive tuple spaces for mobile agent coordination. *Lecture Notes in Computer Science*, pages 237–248, 1998.

[18] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[19] G. Canfora, M. Di Penta, R. Esposito, and M-L. Villani. A Framework for QoS-Aware Binding and Re-binding of Composite Web Services. *Journal of Systems and Software*, 81:1754–1769, 2008.

[20] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola. Qos-Driven Runtime Adaptation of Service Oriented Architectures. In *Proceedings of the the 7th ACM SIGSOFT ESEC/FSE*, pages 131–140, 2009.

[21] A. Carzaniga and A.L. Wolf. Content-based Networking: A New Communication Infrastructure. *Lecture Notes in Computer Science*, pages 59–68, 2002.

[22] Matteo Ceriotti, Amy L. Murphy, and Gian Pietro Picco. Data sharing vs. message passing: Synergy or incompatibility?: An implementation-driven case study. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 100–107, New York, NY, USA, 2008. ACM.

[23] M. Colombo, E. Di Nitto, and M. Mauri. SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In *Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC)*, pages 191–202, 2006.

[24] D. Fensel and F. Fischer and J. Kopecky and R. Krummenacher and D. Lambert and T. Vitvar. WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web (W3C Member Submission). Technical report, W3C, 2010. http://www.w3.org/Submission/WSMO-Lite/.

[25] DARPA. The DARPA Agent Markup Language. Technical report, DARPA. http://www.ai.sri.com/daml/services/owl-s/.

[26] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services. In *Proceedings of VLDB*, 2004.

[27] S. Dustdar and M. Treiber. A View Based Analysis on Web Service Registries. *Distributed and Parallel Databases*, 18(2):147–171, 2005.

[28] A. Dunkel et al. Easy Programming of Integrated Wireless Sebsor Networks - D-1.1: Application and Programming Survey. Technical report, makeSense Project Deliverable, 2010.

[29] Eugster, Patrick Th. and Felber, Pascal A. and Guerraoui, Rachid and Kermarrec, Anne-Marie. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[30] R. Fielding and R. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Interner Technology*, 2(2):115–150, 2002.

[31] Freeman, E. and Arnold, K. and Hupfer, S. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd. Essex, UK, UK, 1999.

[32] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[33] Paul Grace, Gordon S. Blair, and Sam Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):2–14, 2005.

[34] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services. *IEEE Transactions on Services Computing*, 3:223–235, 2010.

[35] Sylvain Hallé and Tevfik Bultan. Realizability analysis for message-based interactions using shared-state projections. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 27–36, 2010.

[36] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[37] Valérie Issarny, Amel Bennaceur, and Yérom-David Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *SFM*, pages 217–255, 2011.

[38] Valérie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A Perspective on the Future of Middleware-based Software Engineering. In *Proceedings of FOSE 2007*, 2007.

[39] J. Kopecky and T. Vitvar. microWSMO:Semantic description of RESTFUL Services (WSMO working draft). Technical report, WSMO, 2008. http://www.wsmo.org/TR/d38/v0.1/.

[40] F. Jammes and H. Smit. Service-Oriented Paradigms in Industrial Automation. *IEEE Transactions on Industrial Informatics*, 1(1):62 – 70, 2005.

[41] Jos de Bruijn et al. WSMO: Web Service Modeling Ontology (W3C Member Submission). Technical report, W3C, 2005. http://www.w3.org/Submission/WSMO.

[42] K Gomadam and A.Ranabahu and A. Sheth. SA-REST: Semantic Annotation of Web Resources (W3C Member Submission). Technical report, W3C, 2010. http://www.w3.org/Submission/SA-REST/.

[43] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[44] B. Liskov and J.M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 16(6):1811–1841, 1994.

[45] Jeff Magee and Jeff Kramer. *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006.

[46] Annapaola Marconi, Marco Pistore, and Paolo Traverso. Automated composition of web services: the astro approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

[47] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transaction on Software Engineering*, 26(1):70–93, 2000.

[48] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of ICSE*, 2000.

[49] Tarek Melliti, Pascal Poizat, and Sonia Ben Mokhtar. Distributed behavioural adaptation for the automatic composition of semantic services. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 146–162, 2008.

[50] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[51] Monson-Haefel, R. and Chappell, D. *Java Message Service*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2000.

[52] Murphy, A.L. and Picco, G.P. and Roman, G.C. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):328, 2006.

[53] NEXOF-RA Project. The NEXOF-RA Reference Model v3.0. Technical report, NEXOF-RA Project. http://www.nexofra.eu/sites/default/files/D6.3_v1.0.pdf.

[54] Elisabetta Di Nitto and David S. Rosenblum. Exploiting adls to specify architectural styles induced by middleware infrastructures. In *Proceedings of ICSE*, 1999.

[55] OASIS. Web Services Quality Model. Technical report, OASIS, 2005. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsqm.

[56] OASIS. Web Services Business Process Execution Language Version 2. Technical report, OASIS, 2007. http://docs.oasis-open.org/wsbpel/.

[57] S. Oundhakar, K. Verma., K.Sivashanugam, A. Sheth, and J. Miller. Discovery of Web Services in a Multi-Ontology and Federated Registry Environment. *International Journal of Web Services Research*, 1(3):1–32, 2005.

[58] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, July 2007.

[59] D. Parnas. On the Criteria for Decomposing To Be Used for Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[60] Jyotishman Pathak, Robyn Lutz, and Vasant Honavar. Moscoe: An approach for composing web services through iterative reformulation of functional specifications. *International Journal on Artificial Intelligence Tools*, 17:109–138, 2008.

[61] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, pages 805–814, 2008.

[62] Peter Pietzuch, David Eyers, Samuel Kounev, and Brian Shand. Towards a common api for publish/subscribe. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 152–157, New York, NY, USA, 2007. ACM.

[63] B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny Web Services for Sensor Device Interoperability. In *Proceedings of the 7th IEEE International Conference on Information Processing in Sensor Networks (ISPN)*, pages 567–568.

[64] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25:206–230, January 1987.

[65] M. Rambold, H. Kasinger, F. Lautenbacher, and B. Bauer. Towards Autonomic Service Discovery A Survey and Comparison. In *Proceedings of the IEEE International Conference on Service Computing (SCC)*, pages 192–201, 2009.

[66] L. Richardson and S Ruby. *RESTful Web Services*. O'Reilly, 2007.

[67] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.

[68] Antony Rowstron. Wcl: A co-ordination language for geographically distributed agents. *World Wide Web*, 1:167–179, 1998. 10.1023/A:1019263731139.

[69] Antony I. T. Rowstron and Alan Wood. Solving the linda multiple rd problem. In *Proceedings of the First International Conference on Coordination Languages and Models*, COORDINATION '96, pages 357–367, London, UK, 1996. Springer-Verlag.

[70] Gwen Salaün. Generation of service wrapper protocols from choreography specifications. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 313–322, 2008.

[71] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *Proceedings of ICSE*, 2003.

[72] Jianwen Su, Tevfik Bultan, Xiang Fu, and Xiangpeng Zhao. Towards a theory of web service choreographies. In *WS-FM*, pages 1–16, 2007.

[73] Dirk Taubner. *Finite representations of CCS and TCSP programs by automata and Petri nets*. Springer-Verlag New York, Inc., 1989.

[74] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software architecture : foundations, theory, and practice*. Hoboken (N.J.) : Wiley, 2009.

[75] CHOReOS Project Team. CHOReOS State of the Art, Baseline and Beyond - Public Project deliverable D1.1, December 2010.

[76] CHOReOS Project Team. CHOReOS dynamic development model definition - Public Project deliverable D2.1, September 2011.

[77] CHOReOS Project Team. CHOReOS Perspective on the Future Internet and Initial Conceptual Model - Public Project deliverable D1.2, March 2011.

[78] Massimo Tivoli and Paola Inverardi. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program.*, 71:181–212, May 2008.

[79] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13:37–85, January 2004.

[80] R. J. van Glabbeek. Notes on the methodology of ccs and csp. *Theor. Comput. Sci.*, 177:329–349, 1997.

[81] W3C. OWL-S: Semantic Markup for Web Services. Technical report, W3C. http://www.w3.org/Submission/OWL-S/.

[82] W3C. Semantic Annotations for WSDL and XML Schema. Technical report, W3C. http://www.w3.org/2002/ws/sawsdl/spec/.

[83] W3C. Web Services Architecture. Technical report, W3C. http://www.w3c.org/TR/ws-arch.

[84] W3C. Simple Object Access Protocol (SOAP) v1.2. Technical report, W3C, 2002. http://www.w3c.org/TR/soap12-part0.

[85] W3C. Web Services Conversation Language (WSCL). Technical report, W3C, 2002. http://www.w3.org/TR/wscl10/.

[86] W3C. Web Services Dynamic Discovery. Technical report, W3C, 2005. http://schemas.xmlsoap.org/ws/2005/04/discovery/.

[87] W3C. Web Services Eventing. Technical report, W3C, 2006. http://www.w3.org/Submission/WS-Eventing/.

[88] W3C. Web Application Description Language. Technical report, W3C, 2009. http://www.w3.org/Submission/wadl/.

[89] W3C. Unified Service Description Language Incubator Group Charter. Technical report, W3C, 2010. http://www.w3.org/2005/Incubator/usdl/.

[90] R. H. Weber and R. Weber. *Internet of Things*. Springer, 2010.

[91] Daniel Wutke, Daniel Martin, and Frank Leymann. Facilitating complex web service interactions through a tuplespace binding. In *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'08, pages 275–280, Berlin, Heidelberg, 2008. Springer-Verlag.

[92] J. Yang and M. Papazoglou. Service Components for Managing the Lifecycle of Service Compositions. *Information Systems*, 29(2):97–125, 2004.

[93] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnamam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.